Model-Based Usability Analysis of Safety-Critical Systems: A Formal Methods Framework

A Thesis

Submitted to the Faculty

of

Drexel University

by

Andrew J. Abbate

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy

February 2017

© Copyright 2017
Andrew J. Abbate. All Rights Reserved.

Dedications

This thesis is dedicated to Salvator Baldi (1921—2011)

Acknowledgments

This work was supported by the U.S. Department of Education Graduate Assistance in Areas of National Need (GAANN) award P200A150023. I thank my co-advisors Prof. Amy L. Throckmorton and Prof. Ellen J. Bass for their patience, guidance, and steadfast enthusiasm for my personal and academic growth. I also thank my committee chair Prof. Patricia Shewokis and committee members Prof. Michelle Rogers and Prof. Kurtulus Izzetoglu for their comprehensive feedback on this work and their patience throughout my Ph.D. process. Finally, I thank Jay Patel, John Rozolis, and Dr. Steven Chopski for assisting with data collection.

Table of Contents

List of Tables . . . xvi

List of Figures . . . xviii

Abstract . . . xxviii

1. Introduction . . . 1

1.1 Model-Based Verification of Usability . . . 4

1.2 Knowledge Gaps . . . 9

1.3 Objectives . . . 11

1.3.1 Navigability of Accompanying Documentation . . . 12

1.3.2 Usability of Procedures in Documentation . . . 12

1.3.3 End-User Capabilities to Configure Hardware in the Operational Environment . . . 13

1.3.4 End-User Interpretation of the Interface . . . 14

1.3.5 Continuous Actuator Dynamics . . . 15

1.3.6 The Integrated Human-System Interface . . . 15

1.4 Contributions . . . 15

1.4.1 A Formal Approach to Documentation: Modeling, Specification, and Verification of Navigability . . . 16

1.4.2 A Formal Approach to Documentation: Modeling, Specification, and Verification of Procedure Usability . . . 17

1.4.3 A Formal Approach to Hardware Configurability: Modeling, Specification, and Verification of Gibsonian Affordance . . . 18

1.4.4 A Formal Approach to Interface Interpretation: Modeling, Specification, and Verification of Signifiers . . . 20

1.4.5 A Formal Approach to Controlled Actuators: Modeling of Continuous Device Dynamics Derived from Spreadsheet Data . . . 21

1.5 An Integrated Framework for Verifying Safety-Critical, Human-Interactive System Usability . . . 22

2. Conceptual Background . . . 26

2.1 Human Factors and Safety-Critical Systems 26
2.1.1 Model-Based Design 27
2.2 Formal Methods 28
2.3 Formal Methods in Human-Interactive Systems 30
2.4 Modeling Methodologies 32
2.4.1 Formalisms 32
2.4.2 Modeling Techniques 34
2.4.2.1 Discrete Device Modeling Techniques 35
2.4.2.2 Continuous Device Modeling Techniques 36
2.4.2.3 Task Modeling Techniques 37
2.4.2.4 Human-System Interaction Encoding Techniques 38
2.4.3 Tool Support for Developing Formal Models 39
2.5 Verification Methodologies 39
2.5.1 Specifications 39
2.5.2 Model Checking Techniques 40
3. Methodologies and Apparatus 42
3.1 Formalisms 42
3.2 Model Checking System 44
3.2.1 SAL Contexts 44
3.2.1.1 The Module Construct 45
3.2.2 Relational Abstraction of Differential Equations in SAL 47
3.2.3 Verification Methodologies 50
3.2.3.1 The SAL Theorem Construct 50
3.2.3.2 Symbolic Model Checking 52
3.2.3.3 Witness Model Checking 53
3.2.3.4 Bounded Model Checking 53
3.2.3.5 Infinite-Bounded Model Checking 53

3.2.4 Verification Apparatus 54
3.3 Encoding Tools 54
4. A Formal Approach to Documentation: Modeling, Specification, and Verification of Navigability 56
4.1 Requirements of a Formalism for Representing Documentation Navigation 59
4.2 Representing Documentation Navigation Formally 60
4.2.1 Formalism 61
4.2.1.1 Documentation Schema 61
4.2.1.2 Navigation Schema 62
4.2.2 Modeling Technique 63
4.2.3 Specifications 64
4.2.4 Modeling Checking Technique 65
4.3 Case Study: A Medical Device PDF User Manual 66
4.3.1 The Formal Model 67
4.3.2 Specifications 69
4.3.2.1 Weak Page Connectedness 69
4.3.2.2 Weak Navigation Completeness 70
4.3.2.3 Cross-Reference Reversibility 70
4.4 Verification 71
4.5 Discussion 72
4.5.1 Methodological Considerations 73
4.5.2 Future Work 73
5. A Formal Approach to Documentation: Modeling, Specification, and Verification of Procedures 75
5.1 Modeling Methodology: Representing Procedures in Documentation Formally 77
5.1.1 Task Modeling Technique 79
5.1.2 Device Modeling Technique 82
5.1.3 System Model Composition 84

5.2 Verification Methodology 84
5.3 Case Study: Left Ventricular Assist Device Alarm Troubleshooting 88
5.3.1 The Device 88
5.3.2 The Draft Manual 90
5.3.3 Formal Task Model 91
5.3.4 Translating from XML to SAL 103
5.3.5 Formal Device Model 104
5.3.6 Verification 109
5.4 Results 111
5.4.1 Encoding of Written Procedure in EOFM-XML 111
5.4.2 Encoding The Device Model 112
5.4.3 Formal Verification of Time-Efficiency Specifications 114
5.5 Discussion 115
5.5.1 Methodological Considerations of the Modeling Approach 117
5.5.2 Methodological Considerations of the Verification Approach 118
5.5.3 Future Work 119
6. A Formal Approach to Hardware Configurability: Modeling, Specification, and Verification of Gibsonian Affordance 121
6.1 Affordance Formalisms 123
6.1.1 Shaw and Turvey [1] 123
6.1.2 Wells [2] 124
6.1.3 Turvey [3] 125
6.1.4 Stoffregen [4] 126
6.1.5 Thiruvengada and Rothrock [5] 126
6.1.6 Greeno [6] 127
6.1.7 Chemero [7] 128
6.1.8 Lenarčič and Winter [8] 128
6.1.9 Warren [9] 129

6.2 Representing Affordances Formally 130

6.2.1 Requirements of a Modeling Technique 130

6.2.2 Requirements of an Encoding Tool 132

6.3 The CAVEMEN Approach 136

6.3.1 The Root Node 138

6.3.2 Model Objects, Subobjects, and Atomic Objects 138

6.3.3 Affordance 138

6.3.4 Human Operator 139

6.3.5 Relation 139

6.3.6 Component, Subcomponent, and Atom Component 141

6.3.7 Ability 141

6.3.8 Translatable, Positionable and Orientable 142

6.3.9 Translation to SAL 143

6.3.9.1 Automatically Generated Types 144

6.3.9.2 Affordance Module Variables 145

6.3.10 HES Module Variables 146

6.3.11 Affordance Module Definitions 146

6.3.12 HES Modeling Technique 149

6.3.13 Module Composition 151

6.3.14 Specifications 151

6.3.15 Model Checking Technique 153

6.4 Case Study 154

6.4.1 System Description 155

6.4.2 Event Description 156

6.4.3 CAVEMEN-XML Model 158

6.4.3.1 LVLeadConnectableToRVPort 159

6.4.3.2 LVLeadConnectableToLVPort 161

6.4.3.3 RVLeadConnectableToRVPort 161

6.4.4 SAL Representation 161

6.4.5 Specifications 163

6.4.6 Verification 165

6.5 Scalability 166

6.6 Discussion 168

6.6.1 Methodological Considerations 169

6.6.2 Future Work 170

7. A Formal Approach to Interface Interpretation: Modeling, Specification, and Verification of Signifiers 172

7.1 Representing Signifiers Formally 175

7.1.1 Requirements of a Signifier Formalism 176

7.1.2 Requirements of an Encoding Tool 178

7.2 The BIGSIS Approach 179

7.2.1 Representing Signifiers Formally: The BIGSIS Formalism and Modeling Technique 181

7.2.2 BIGSIS Formalism Semantics 184

7.2.2.1 Values Schema 185

7.2.2.2 Properties_signify Schema 186

7.2.2.3 Signifiers Schema 187

7.2.2.4 Next_state_signifiers Schema 191

7.2.2.5 Outputs Schema 192

7.2.3 Specifications 194

7.2.3.1 Signifier Consistency 194

7.2.3.2 Signifier Redundancy 195

7.2.3.3 Signifier Completeness 197

7.2.3.4 Constrained Specifications 197

7.2.4 BIGSIS-XML 198

7.2.4.1 Signified Functions and Meanings 198

7.2.4.2 Signifier Properties . . . 199
7.2.4.3 Property Documentation . . . 200
7.2.4.4 Properties Operating as Signifiers . . . 200
7.2.5 Formal Model Translation . . . 204
7.2.5.1 SAL Representation of *values* Schema . . . 205
7.2.5.2 SAL Representation of *properties_signify* Schema . . . 205
7.2.5.3 SAL Representation of *signifiers* Schema . . . 206
7.2.5.4 SAL Representation of *next_state_signifiers* Schema . . . 208
7.2.5.5 SAL Representation of *outputs* Schema . . . 208
7.2.5.6 System Model Composition . . . 209
7.2.5.7 Automatic Generation of Signifier Specifications . . . 209
7.2.6 Model Checking Technique . . . 211
7.3 Case Study . . . 211
7.3.1 System Description . . . 212
7.3.2 BIGSIS Model . . . 213
7.3.3 Device Model . . . 216
7.3.4 Specifications . . . 217
7.3.4.1 Signifier Consistency . . . 218
7.3.4.2 Signifier Redundancy . . . 219
7.3.4.3 Signifier Completeness . . . 219
7.3.5 Verification . . . 220
7.4 Scalability . . . 221
7.5 Discussion . . . 224
7.5.1 The BIGSIS Formalism and Modeling Technique . . . 224
7.5.2 Signifier Specifications . . . 225
7.5.3 Encoding Tool: BIGSIS-XML and Translator . . . 226
7.5.4 Medical Device Case Study . . . 227

7.5.5 Scalability 228

8. A Formal Approach to Controlled Actuators: Modeling of Continuous Device Dynamics Derived from Spreadsheet Data 229

8.1 Modeling Methodology: Representing Testing Data Formally 231

8.1.0.1 Formal Semantics of Translated SAL Models 232

8.1.1 Verification Methodology 235

8.2 Case Study 236

8.2.1 The Device 236

8.2.2 Testing Data 237

8.2.3 Translation 238

8.2.4 Additional SAL Model Infrastructure 239

8.2.5 Verification 242

8.3 Scalability 243

8.4 Discussion 245

8.4.1 Methodological Considerations 246

8.4.2 Future Work 247

9. An Integrated Framework for Verifying Safety-Critical, Human-Interactive System Usability 248

9.1 Modeling Methodology: Integrated Framework Architecture 249

9.2 Verification Methodology: Integrated Specifications and Model Checking Technique . . 257

9.2.1 Specifications 257

9.2.2 Model Checking Technique 269

9.3 Case Study 270

9.3.1 Configurable Hardware 271

9.3.2 Modified Controller 273

9.3.3 Modified Documentation 275

9.4 Integrated Framework Model 280

9.5 Documentation Navigation 282

9.6 Task Models . . . 282
9.6.1 Pump Speed Adjustment . . . 283
9.6.2 Pump Stopped Alarm Troubleshooting . . . 285
9.7 Affordances . . . 292
9.7.1 CAVEMEN-XML Model . . . 294
9.7.1.1 Modelobject, Subobject, and Atomicobject Nodes . . . 294
9.7.1.2 Affordance Nodes . . . 298
9.7.1.3 Affordances Involving the Permanently Attached Connector . . . 299
9.7.1.4 Affordances Involving Cable Disconnectability . . . 301
9.7.1.5 Affordances Involving Cable Connectability . . . 304
9.7.2 HES Module . . . 311
9.8 Device . . . 312
9.8.1 End User-Device Interaction . . . 312
9.8.2 Display/Control Logic . . . 313
9.8.3 Plant and Constraints . . . 316
9.9 Signifiers . . . 317
9.9.1 BIGSIS-XML Model . . . 318
9.9.2 BIGSIS-SAL Model . . . 320
9.9.2.1 End-User Descriptions . . . 321
9.9.2.2 Documentation Channel . . . 323
9.10 System Model Composition . . . 325
9.11 Specifications . . . 326
9.12 Verification . . . 332
9.12.1 Results . . . 332
9.13 Discussion . . . 341
9.13.1 Methodological Considerations . . . 343
10. Contributions and Future Work . . . 345

10.1 Specific Contributions 350

10.2 Future Work 351

10.2.1 Practical Applications 351

10.2.1.1 Informing Improved Usability Standards 351

10.2.1.2 Evaluating Usability of EAP Medical Devices 352

10.2.2 Future Development of the Framework 352

10.2.2.1 Extensions to the Modeling and Verification Methodologies 352

10.2.2.2 Case Study Implementations 354

10.2.2.3 Tool Support 355

10.3 Conclusion 357

List of References 358

Appendix A: List of Symbols 373

Appendix B: Tool Support Listing 384

Appendix C: Listing of Additional Encoding Techniques 385

C.1 HES Module Syntax for Spatial Relation Transitions Using Stoffregen's Formalism . . 385

C.2 HES Module Syntax for Initializing Human Capabilities 385

C.3 Abstracting Learned Documentation Content in the Integrated Framework 386

Appendix D: Chapter 4 Code Listing 388

D.1 SAL Model of User Manual Navigation (Section 4.3.1) 388

Appendix E: Chapter 5 Code Listing 389

E.1 EOFM-XML Formal Task Model (Section 5.3.3) 389

E.2 SAL Model of Human-Device Interaction (Section 5.3.5) 392

Appendix F: Chapter 6 Code Listing 397

F.1 XML Code 397

F.2 SAL Code 398

Appendix G: Chapter 7 Code Listing 402

G.1 XML Code 402

G.2 SAL Code 403
G.2.1 BIGSIS-SAL Model 403
G.2.2 Device Model 406
Appendix H: Chapter 9 Code Listing 408
H.1 XML Code 408
H.1.1 Task Models 408
H.1.2 Affordance Model 411
H.1.3 Signifier Model 416
H.2 SAL Code 417
H.2.1 Documentation Navigation 417
H.2.2 Task Model 418
H.2.3 Affordance Model 430
H.2.4 Signifier Model 437
H.2.5 Discrete Device 441
H.2.6 Continuous Device Model 445
H.3 Specifications 448
H.3.1 Accuracy and Understandability 448
H.3.2 Accuracy and Error Tolerance 448
H.3.3 Accuracy and Time Efficiency 448
H.3.4 Accuracy and Completeness 448
H.3.5 Understandability and Error Tolerance 448
H.3.6 Understandability and Time Efficiency 448
H.3.7 Error Tolerance and Time Efficiency 448
H.3.8 Error Tolerance and Completeness 448
H.3.9 Understandability and Completeness 449
H.3.10 Time Efficiency and Completeness 449
Vita 450

List of Tables

1.1 Examples of possible end user-interface, controlled actuator-interface, and integrated user-actuator-interface interactions that could emerge in safety-critical systems. Designed elements include interface components and controlled actuators, all of which must be usable in the operational environment. Interactions between documentation and controlled actuators are not considered in this work (denoted by blank cell) 2

1.2 Broad overview of the scope covered by extant modeling and verification methodologies (works referenced in cells), areas addressed in this research (chapters referenced in cells), and areas that should be explored in future work. Interactions between documentation and controlled actuators are not considered in this work 16

1.3 Descriptions of temporal logic specifications that can be encoded in LTL for an integrated framework model 25

2.1 Boolean operators, logical connectives, and example propositions that can be incorporated within temporal logic specifications. ϕ and ψ are hypothetical formal model variables that can be valued 0 or 1 29

2.2 Propositions from Table 2.1 augmented with temporal operators and path quantifiers that are needed in temporal logic specifications. LTL specifications have temporal operators. CTL specifications have path quantifiers and temporal operators 30

4.1 Computational tree logic (CTL) representation of page reachability specifications 65

4.2 Case study model checking results 71

5.1 Decomposition Operators [10] 78

5.2 Results of encoding the formal device model 113

5.3 Case study model checking results 114

6.1 Topology attribute values. All topological relations are mutually exclusive 140

6.2 Direction attribute values 141

6.3 Case study model checking results 165

6.4 Results of scalability evaluation 168

7.1 Constrained set of BIGSIS-XML perceivable property nodes (corresponding to P_i of the BIGSIS formalism). Exemplars of descriptions (corresponding to D_i of the BIGSIS formalism) and descriptive words, terms, or phrases (corresponding to d_0, ..., d_n of the BIGSIS formalism) are shown in the second and third columns 202

7.2 Control knob settings, programmed speeds, and approximate power supplied by the controller discoverable within accompanying documentation 213

7.3 Case study model checking results 220

7.4 Results of scalability evaluation. "—" indicates that model checking failed 224

8.1 Tabulated representation of hypothetical testing data in a file named *example.csv* 234

8.2 Testing data for the case study device at five rotational speeds. "•" indicates no data collected, corresponding to empty cells of the spreadsheet 238

8.3 LTL Specifications and model checking verification times 242

8.4 Model checking results for scalability tests. Both tests for a single model are reported in the same row. "—" in the verification time column indicates that model checking failed 245

9.1 Control knob settings, programmed speeds, and approximate power supplied by the controller leveraged from simulation data (Chapter 8) 274

9.2 Model checking results 333

10.1 Human-system interface elements that must be modeled and usability measures that must be verified in safety-critical systems. "✓ " denotes that an area of methodological support has been addressed by other researchers; "★" denotes that an area was addressed this work; and '—" denotes and area must be addressed in future work 345

List of Figures

1.1 A subset of specifications that have proven useful within safety-critical system analyses, organized as they relate to accuracy (1), understandability (2), error tolerance (3), time efficiency (4), and completeness (5). Numbers and letters serve as a reference for the integrated frameworks in which they have been utilized 5

1.2 Graphical representation of the integrated framework. Elements of the target system are color-coded to identify corresponding framework models 23

2.1 Graphical FSM representation of a household range. Variables are boldfaced and italicized within square-edge rectangles. State values are italicized within rounded-edge rectangles. Initial states are indicated by curved arrows having no label and a filled circle. Guarded transitions are indicated by labeled arrows. Unguarded transitions are indicated by unlabeled arrows. (a) The dial having an initial state of 0. It can remain zero or transition to 1. Subsequent states can remain unchanged, transition up-one, or transition down-one. (b) The burner light having an initial state of *OFF*. It remains *OFF* if the dial is set to 0, and it transitions to *ON* if the dial is positive. When the indicator light is in the *ON* state, it remains *ON* if the dial is positive, and it transitions to *OFF* if the dial is set to 0. 35

3.1 A generic XML document with corresponding graphical and textual representations. (a) The graphical representation of XML code utilized throughout this document. Nodes are contained within square-edge rectangles. Attributes are contained within rounded-edge rectangles. Text content is contained within parallelograms. Arrows point from parent to child nodes. Sibling nodes are grouped within shapes. Bold text in parentheses is added to aid in identifying XML line numbers in (b) that correspond to each graphically represented node. (b) XML code represented graphically in (a) 55

4.1 A generic example depicting the structure and function of hyperlinking in the table of contents. Clicking on the section number, 5, navigates the user to page-200. Additionally, clicking on a page number navigates the user to that page 57

4.2 Generic examples depicting the structure and function of hyperlinking in the PDF user manual. (a) Index: Clicking on a page number navigates the user to that page. (b) Main body of the user manual: Clicking "*Fig. 45*" navigates the user to the page containing Fig. 45. Clicking "*Battery Holder* on page 121" navigates the user to page 121 57

4.3 CTL specifications and path trees representing paths through different possible designs of an aircraft QRH 66

4.4 A fragment of the link report generated by the EverMap LLC *AutoBookmark* plugin tool [11] 68

4.5 Tree representations of the witnesses provided in verification reports generated by SAL-WMC: (a) invalid result for *weak page connectedness*. (b) Valid result for *weak navigation completeness*. (c) Invalid result for *cross-reference reversibility* 72

5.1 Graphical depiction of a generic EOFM-XML activity. "Decomposition operator" can be replaced with any of the keywords shown in Table 5.1. "Precondition" and "Completion condition" can be replaced with valued input variables representing states of the device or operational environment . . . 79

5.2 Generalizable, graphical representation of the task modeling technique for representing a procedure in accompanying documentation. Annotations in italic text are not part of EOFM-XML's formal semantics. Letters a–f are added for reference in the outlined description of this technique . . . 80

5.3 Rendering of system components involved in the troubleshooting procedure are listed as "Component Name (Quantity involved in the draft manual procedure and the formal task model)" . . . 90

5.4 Outline form of case study troubleshooting procedure . . . 91

5.5 Visualization of the six main steps of the troubleshooting procedure encoded as ten EOFM sub-activities. The top-level activity *aRespondToPumpStoppedAlarm* represents the entire procedure, while the ten sub-activities represent end-user activities prescribed within the six main steps. A top-level activity precondition specifies that the procedure begins executing when the pump stopped alarm engages. A completion condition specifies that the procedure completes execution when the alarm disengages. The *ord* decomposition operator specifies that all ten sub-activities must execute in order . . . 92

5.6 Visualization of the formal task model representing step 1a . . . 94

5.7 Visualization of the formal task model representing step 1b . . . 96

5.8 Visualization of the formal task model representing step 1c . . . 97

5.9 Visualization of the formal task model representing steps 2 and 3 . . . 99

5.10 Visualization of the formal task model representing step 4 . . . 100

5.11 Visualization of the formal task model representing step 5a . . . 101

5.12 Visualization of the formal task model representing step 5b . . . 102

5.13 Visualization of the formal task model representing step 6a . . . 103

5.14 Visualization of the formal task model representing step 6b . . . 103

5.15 Visual representations of device state transitions encoded in the HDI model. Text written directly above and below arrows define the conditions that must evaluate to *true* for the respective state transition to execute. (a) Cable connections and disconnections. (b) State of the alarm battery cap on the old controller. (c) State of the alarm battery cap on the new controller . . . 108

5.16 Visual representations of device state transitions encoded in the HDI model. Text written directly below arrows states the conditions that must evaluate to *true* for the respective state transition to execute. Variable names are listed in bold italic text within each gray, rounded-edge rectangle. (a) Position of all old, removed components relative to the patient. (b) Attaching red tags (Fig.5.3r) to old, removed components. (c) Status of the connector permanently attached to the heart (Fig. 5.3f). (d) 90° rotations performed on the connector permanently attached to the heart (Fig. 5.3f). (e, f, g) Lights illuminated on the old lithium-ion battery (Fig. 5.3q), new lithium-ion battery (Fig. 5.3q), and lead reserve battery respectively. Each of these variable values can transition to any one of the numbers within white, rounded edge squares 109

5.17 Visualization of the two-step counterexample to *Preparatory action time inefficiency* in the *or_seq* model. Green, rounded-edge rectangles are activities that are executing. The green, square-edge rectangle indicates that *hReassembleBrokenConnector* is valued true in the state violating the specification 115

5.18 Visualization of the two-step counterexample to *Preparatory action time inefficiency* in the *or_seq* model. Green, rounded-edge rectangles are activities that are executing. The green, square-edge rectangle indicates that *hReassembleBrokenConnector* is valued true in the state violating the specification 116

6.1 (a-d) Conceptual representations of two-dimensional topological relationships inspired by the formal language described in [12]: (a) 1 is disjoint to 2, (b) 1 touches 2, (c) 1 covers 2. (d) 1 overlaps 2. (e) A graphical representation of the topological-spatial relations: 2 is to the back of 1, 3 is to the right of 1, and 4 is to the bottom of 1, inspired by the techniques described in [13] and [14]. 134

6.2 A visual representation of positional and rotational movements based on the six degrees of freedom of a rigid body in three dimensional space. Rotational movements are indicated by the arcing arrows. Positional movements are indicated by the straight arrows. 135

6.3 Visual representation of the CAVEMEN-XML grammar. In a–n, rectangles are nodes, rounded-edge rectangles are attributes, and arrows point to child nodes. Sibling nodes are horizontally adjacent. Required nodes and attributes are outlined in dotted red lines. For sibling nodes, a blue dashed outline indicates that at least one of either is required. Groupings of nodes within shapes correspond to the subsections describing them in Section 6.3. 137

6.4 A graphical representation of an example CAVEMEN-XML *affordance* node and translated SAL model infrastructure for three extant affordance formalisms. (a) An example CAVEMEN-XML *affordance* node. Arrows inside the box point from parent nodes to child nodes. Arrows outside the specify different *formalism* attribute values and point to corresponding SAL model infrastructure. Nodes are colored to identify corresponding SAL code: in b–d, variable declarations are enclosed in color-coded rectangles and value assignments are written in color-coded text. (b) SAL model infrastructure generated for the *formalism* attribute value *stoffregen*. (c) SAL model infrastructure generated for the *formalism* attribute value *greeno*. (d) SAL model infrastructure generated for the *formalism* attribute value *chemero* 148

6.5 (a–h) Graphical rendering of case study device components. Red rectangles indicate proximal tip and input port surfaces that must touch to establish a connection. (a) Pulse generator. (b) LV input port. (c) LV set screw. (d) RA input port. (e) RA set screw (f) RV input port. (e) RV set screw. (h) Lead segments. Arrows indicate continuation of middle segments and implanted positions of distal tips. Proximal tip segments begin at the top edges of white rectangles containing letters i–k. (i) LV lead proximal tip. (j) RA lead proximal tip. (k) RV lead proximal tip. (m) Lines 1–12 of instantiated CAVEMEN-XML model. 155

6.6 Graphical rendering of HES entities in a configuration satisfying *relation* nodes and attributes. Labeled axes in (a) show the surgeon's visual perspective with respect to surfaces of the pulse generator, all three ports, and all three set screws. Labeled axes in (b) show the surgeon's visual perspective with respect to all surfaces of the LV lead proximal tip. RA and RV leads are perceived in the same way. (a) Surgeon (not shown) disjoint to front of RV port. (b) Surgeon (not shown) disjoint to top, bottom and front of LV lead proximal tip. (c) LV lead proximal tip not covering back of RV port. (d) RV set screw touching right of pulse generator, and RA set screw disjoint to right of pulse generator for visual comparison. (e) RA lead proximal tip disjoint to front of RV port. (f) RV lead proximal tip disjoint to front of RV port. (g) RA lead proximal tip and RV lead proximal tip both disjoint to front of LV lead proximal tip (tabletop surface not modeled). (h) Lines 13–51 of CAVEMEN-XML model. 157

6.7 Rendering of the three-step trace leading up to a safe state in which the surgeon connects to RV lead to the RV port. (a) LV lead connectable to the LV port and the RV lead connectable to RV port. (b) RV lead connectability to the RV port is in the process of being actualized. (c) RV lead connected to the RV port 166

6.8 Rendering of the three-step trace leading up to an unsafe state in which the surgeon erroneously connects to LV lead to the RV port. (a) LV lead connectable to the LV port and RV lead connectable to the RV port. (b) LV lead connectability to the RV port is in the process of being actualized. (c) LV lead connected to RV port 166

7.1 Visual representations of one relation function and one explanation function for a hypothetical smoke detector interface (a) and its accompanying documentation (b). Underlined text above each oval identifies *signifiers* schema variables instantiated for the smoke detector. Italic words within rectangles are a set of end-user description words from a set of audible pattern descriptions (corresponding to $\mathbb{F}_1 D_i$)). Italic words within ovals are a set of meaning words from a category of signified alarm meanings (corresponding to $\mathbb{F}_1 S_i$)). Arrows pointing from one description word to one meaning word represent the input/output behavior of each function (a) The relation function for the smoke detector's audible pattern. (b) The explanation function for accompanying documentation explaining what is signified by the smoke detector's audible pattern 191

7.2 Visual representation of the BIGSIS-XML grammar. Square-edge rectangles are nodes. Smaller, round-edge rectangles are attributes. Parallelograms are text content. Arrows point from parent to child nodes. Boldface headings aid in identifying groups of similar perceivable properties and are not part of the grammar (a) The root node *bigsis*. (b–d) Direct children of *bigsis*. (e) Direct children of *signifier-properties* and *property-documentation*. The formal semantics of nodes f–g are leveraged from the BIGSIS formalism. (f) Visual signifiers. (g) Orientation-dependent visual signifiers. (h) Audible signifiers. (i) Haptic signifiers 199

7.3 Graphical representation of signifier properties considered in the BIGSIS formalism. Boldface headings describe what is common among properties operating as signifiers. Signifier names are italicized within rectangles. Signifiers have one description (italicized within rounded, dashed edge rectangle) and one or more categories of signified function or meaning (italicized within rounded, solid edge rectangle) Prefix *o* stands for orientation-dependent. For time-variant properties of pattern operating through different channels, prefix *v* stands for visual, *a* for audible and *h* for haptic. (a) Visual signifiers. (b) Orientation dependent visual signifiers. (c) Audible signifiers. (d) Haptic signifiers . . . 201

7.4 Visual representation of SAL code generated from a generic BIGSIS-XML instantiation. Arrows and bold-italic labels indicate which BIGSIS-XML nodes are parsed by the translator to represent SAL code. (a) Graphical representation of instantiated BIGSIS-XML. (b) Formal signifier model SAL code generated by the translator. (c–g) Portions of the translated SAL formal model corresponding to BIGSIS formalism schemas. (h) Four of the eight automatically generated LTL signifier specifications. Not shown: audible signifier consistency, documentation signifier consistency, visual and documentation partial signifier redundancy, audible and documentation partial signifier redundancy 207

7.5 (a) Labeled diagram of the case study system's battery-powered controller appearing within accompanying documentation. Letters b–d and dashed red boxes are added to identify the three components considered in the case study. (b) Power indicator lights (numbered 3–12) and high power alarm (numbered 13). (c) Pump stopped alarm. (d) Speed setting knob. (e) Graphical rendering of the controller. The speed setting is four and the power indicator lights 8–9 are illuminated green. (f) High power alarm engaged: the number 13 illuminates amber and a loud, continuous alarm sounds. (g) Pump stopped alarm engaged: the octagon shaped light illuminates red and a loud, continuous alarm sounds 212

7.6 Graphical representation of the device model. Arrows connecting rounded-edge rectangles represent transitions. Curved arrows with filled circle represent initial states. (a) device model infrastructure representing the system's alarms. The variable *Alarm* is an output operating as an input to the *signifiers* module. (b) device model infrastructure representing human-system interaction. The variable *Action* is an output operating as an input to the SAL *signifiers* module. (c) Label of the power indicator lights. (d) Color of the power indicator lights. (e) Color of the pump stopped alarm light. (f) Label of the speed setting knob 217

7.7 Visualization of case study counterexample to *visual consistency*. Variables and values are listed in italic text within rectangles. Arrows point from variables to interface components. Conflicting meanings are listed in boldface white text within black rectangles . 220

7.8 Visualization of case study counterexample to *PowerSupplied visual audible redundancy*. Variables and values are listed in italic text within rectangles. Arrows point from variables to interface components. Conflicting meanings are listed in boldface white text within black rectangles 221

7.9 Visualization of case study counterexample to *PumpSpeed visual documented redundancy*. Variables and values are listed in italic text within rectangles. Arrows point from variables to interface components. Conflicting meanings are listed in boldface white text within black rectangles 222

8.1 (a) Graphical representation of tabulated data having rows, columns, and slices. (b) Graphical representation of blood pump testing data having two kinds of measurements (flow rate on the x-axis and power on the y-axis) and five slices (rotational speed represented by each line) . . . 230

8.2 A graphical representation of continuous model architecture employed within model checking analyses. (a) A continuous device model having synchronously composed *plant* and *constraints* models. (b) One or more LTL specifications encoded using continuous device model outputs, where `Output_i = value` is a generic *plant* model state that should always hold . . . 236

8.3 Graphical rendering of the case study device. Reprinted from [15] . . . 237

9.1 Diagrammatic representation of the model architecture employed in the holistic human-system interface framework. Outgoing arrows direct output variables from one model as inputs to another. Labels within each arrow identify what variables are exchanged and what behaviors are controlled. Arrows connecting to shapes labeled *xor* indicate that the analyst must implement the framework using exactly one set of variables identified in the labels of incoming arrows (j) or outgoing arrow (k). Arrows connecting to the shape labeled *and* indicate that the analyst must implement the framework using both sets of variables connected to the shape. Arrows having dotted lines indicate that the use of a page number output of the *documentation navigation* model is optional . . . 253

9.2 Labeled diagram of case study configurable hardware components. This figure appears on pages 13 and 14 of the modified patient handbook to aid end users in identifying components involved in the pump stopped alarm troubleshooting procedure . . . 272

9.3 Graphical rendering of the case study system's controller. (a) The speed setting number 4 illuminates white and the power indicator light 6 illuminates green. (b) High power alarm engaged: the word "HIGH" illuminates amber and a loud alarm emits the phrase "power too high" with a one-second pause between emissions. (c) Pump stopped alarm engaged while power is still supplied to the pump: the octagon shaped light illuminates red, one power indicator light illuminates green, the speed setting knob light turns off, and a loud alarm emits the phrase "pump stopped" with a one-second pause between emissions. (d) Pump stopped alarm engaged while no power supplied to the pump: the octagon shaped light illuminates red, the speed setting knob light turns off, the power indicator light turns off, and a loud alarm emits the phrase "pump stopped" with a one-second pause between emissions. . . . 273

9.4 Graphical rendering of pages in the draft patient handbook . . . 276

9.5 Outline form of case study troubleshooting procedure . . . 280

9.6 Graphical representation of the task model representing the user manual procedure for adjusting speed setting on the pump's controller . . . 284

9.7 Visualization of the six main steps of the modified troubleshooting procedure encoded as six EOFM sub-activities. The top-level activity *aRespondToPumpStoppedAlarm* represents the entire procedure, while the six sub-activities represent end-user activities prescribed within the six main steps. A top-level activity precondition specifies that the procedure begins executing when the pump stopped alarm engages. A completion condition specifies that the procedure completes execution when the alarm disengages. The *ord* decomposition operator specifies that all six sub-activities must execute in order . . . 285

9.8 Visualization of the formal task model representing step-1 286

9.9 Visualization of the formal task model representing step-2 286

9.10 Visualization of the formal task model representing step-3 288

9.11 Visualization of the formal task model representing step-4. The activities *aCallEmergencyNumber* and *aCallEmergency* represent the same task. This task appears twice because the first instance, *aCallEmergencyNumber*, has a precondition (*iNewLiBattLights* < *5*) while the second instance, *aCallEmergency*, has no precondition 289

9.12 Visualization of the formal task model representing step-5 290

9.13 Visualization of the formal task model representing step-6 292

9.14 Graphical representation of the case study system's old controller, old lithium-ion battery cable, and CAVEMEN-XML nodes representing them. Node names are listed within square-edge rectangles. *Name* attribute values are listed within rounded-edge rectangles. Letters aid in identifying what components correspond to *modelobject* nodes and what parts correspond to *subobject* and *atomicobject* nodes 295

9.15 Graphical representation of the case study system's pump cable, lead battery, and CAVEMEN-XML nodes representing them. Node names are listed within square-edge rectangles. *Name* attribute values are listed within rounded-edge rectangles. Letters aid in identifying what components correspond to *modelobject* nodes and what parts correspond to *subobject* and *atomicobject* nodes 296

9.16 Graphical representation of the case study system's abdominal cable, old lithium-ion battery, Y-cable, and CAVEMEN-XML nodes representing them. Node names are listed within square-edge rectangles. *Name* attribute values are listed within rounded-edge rectangles. Letters aid in identifying what components correspond to *modelobject* nodes and what parts correspond to *subobject* and *atomicobject* nodes 297

9.17 (a) Labeled axes show the pump operator's visual perspective for surfaces of the pump cable (*mPumpCable*). Surfaces are the same for the permanently attached connector (*sConnector*) and connector parts (*aoConnectorPart1*/*aoConnectorPart2*). (b–c) Graphical renderings of configurations satisfying *relation* nodes (*relation* nodes not shown). (b) *aoConnectorPart1* disjoint to *aoConnectorPart2*. (c) *aoConnectorPart1* covering front-side surface of *aoConnectorPart2* 300

9.18 (a–c) Labeled axes show the pump operator's visual perspective for the: (a) abdominal cable (*mAbdominalCable*), (b) pump cable (*mPumpCable*), and (c) old controller (*mOldController*). (d–e) Graphical renderings of configurations satisfying *relation* nodes (*relation* nodes not shown). (d) Abdominal cable input socket (*aoACPumpInput*) covering all surfaces of the pump cable output end (*aoPCControllerOutput*). (e) Old controller pump cable input socket (*aoOCPumpInput*) covering all surfaces of the pump cable output end (*aoPCControllerOutput*) 302

9.19 (a–c) Labeled axes show the pump operator's visual perspective for the: (a) lead battery cable output end (*aoLeadBattControllerOutput*) (b) new lithium-ion battery cable (*mNewLiBattCable*) and its output ends (*aoNBCControllerOutput/aoNBCControllerOutput*), and (c) new controller (*mNewController*). (d) New controller battery cable input socket (*aoNCBatteryInput*) covering all surfaces of the lead battery cable output end (*aoLeadBattControllerOutput*). (e) New controller battery cable input socket (*aoNCBatteryInput*) covering all surfaces of the new lithium-ion battery cable output end (*aoNBCControllerOutput*) 304

9.20 (a–b) Labeled axes show the pump operator's visual perspective for (a) the pump cable output end (*aoPCControllerOutput*) and (b) new controller (*mNewController*). (c) Pump cable output end is disjoint to all components 305

9.21 (a–c) Labeled axes show the pump operator's visual perspective for: (a) the lead battery cable output end (*aoLeadBattControllerOutput*), (b) the new controller (*mNewController*), and (c) the new lithium-ion battery cable output ends (*aoNBCControllerOutput/aoNBCBatteryOutput*) and old lithium-ion battery cable output ends (*aoOBCControllerOutput/aoOBCBatteryOutput*). (d) Lead battery cable output end disjoint to the controller's battery input socket. (e) New (or old) lithium-ion battery cable's controller output end disjoint to the new controller's battery input socket 307

9.22 (a, b) Labeled axes show the pump operator's visual perspective for the: (a) new lithium-ion battery cable output ends (*aoNBCControllerOutput/aoNBCBatteryOutput*) and old lithium-ion battery cable output ends (*aoOBCControllerOutput/aoOBCBatteryOutput*), (b) new lithium-ion battery (*mNewLiIonBattery*) and old lithium-ion battery (*mOldLiIonBattery*). (c) New (or old) lithium-ion battery cable's battery output end disjoint to new controller's battery input socket 310

9.23 Visual representation of transitions encoded in the *end user-device interaction* model. (a) General encoding of the eight cable connection/disconnection transitions involving end-user connection/disconnection actions and connectable/disconnectable affordances. (b) Transitions involving the end-user action of rotating connector parts and the affordance of part rotatability. (c) Transition involving the end-user actions of assembling/disassembling parts of the permanently attached connector and assemblable/disassemblable affordances (d) Transitions to the speed setting knob. The EOFM human action variable *hRotateKnobCounterClockwise* represents the end user rotating the knob in the direction of the curved arrow to increase the speed setting by one. The variable *hRotateKnobClockwise* represents the end user rotating the knob in the opposite direction to decrease the speed setting by 1 314

9.24 Graphical representation of guarded initializations of the *display/control logic* model. Variable names are listed in boldface, italic text within square-edge rectangles. Variable values are listed in italic text within rounded-edge rectangles. Arrow labels in italic text are valued input variables from the *plant* model. (a) Transitions for the variable *iAlarm* representing what alarm is engaged on the controller. (b) Transitions for the variable *iPowerLight* representing what indicator light is illuminated on the controller. The value "*0*" corresponds to no lights illuminated. The values *1–10* correspond to lights having numeric labels 1–10. The value "*11*" corresponds to the light labeled "*HIGH*" 315

9.25 Graphical representation of guarded transitions to end-user descriptions of color. Variable names are listed in boldface, italic text within square-edge rectangles. Variable values are listed in italic text within rounded-edge rectangles. Arrow labels in italic text are valued variables from the *display/control logic* model. (a) End-user description of color for the power indicators. (b) End-user description of color for the speed setting knob. (c) End-user descriptions of color for the pump stopped alarm 321

9.26 Graphical representation of guarded transitions to end-user descriptions of label. Variable names are listed in boldface, italic text within square-edge rectangles. Variable values are listed in italic text within rounded-edge rectangles. Arrow labels in italic text are valued variables from the *display/control logic* or *end user-device interaction* models. (a) End-user description of label for the power indicators. (b) End-user description of label for the speed setting knob 322

9.27 Graphical representation of guarded transitions to end-user descriptions of volume and audible pattern. Variable names are listed in boldface, italic text within square-edge rectangles. Variable values are listed in italic text within rounded-edge rectangles. Arrow labels in italic text are valued variables from the *display/control logic* or *end user-device interaction* models. (a) End-user description of volume for the pump stopped alarm. (b) End-user description of volume for the power indicators. (b) End-user description of audible pattern for the pump stopped alarm. (d) End-user description of volume for the power indicators 324

9.28 Graphical representation of the case study counterexample trace through the *plant* model. Flow rate in liters per minute (LPM) is shown on the x-axis. Power supplied in watts in shown on the y-axis. Colored lines shown the flow rate and power supplied for each speed setting 1–5 (corresponding to 2,000—6,000 RPM). The black circle indicates that the initial and final states of power, speed, and flow, are 0 (variables and values listed inside dashed-box rectangle) 334

9.29 Graphical representation of the case study counterexample trace through the *documentation navigation* model. The curved arrow indicates that the initial state (step-0) is page-2. Straight arrows indicate next-states that were possible in the counterexample trace. Grey circles indicate the pages are along the path. The black circle indicates that the final state is page-13; i.e., *iPage = 13* for all remaining steps of the counterexample trace 334

9.30 Graphical representation of the case study counterexample trace through the pump stopped alarm troubleshooting procedure *task* model. White, rounded-edge rectangles are activities that did not execute. White, square-edge rectangles are human actions that did not execute. Colored, rounded-edge rectangles are activities that executed. Rectangles having the same color are heterarchical. Black, square-edge rectangles are human actions that executed 335

9.31 Graphical representation of *end user-device interaction* and *affordance* model initial states in the case study counterexample. Letters are added for reference in text of Section 9.12.1. In a–d, dashed-line rectangles contain enlarged graphical renderings of initial states for a subset of configurable hardware components. Solid-line rectangles show the corresponding variable value(s) from the end user-device interaction model. In e–g, solid-line rectangles show variable of interest from the *affordance* model) 336

9.32 Graphical representation of *end user-device interaction* and *affordance* model states in intermediate steps of the case study counterexample. Letters are added for reference in text of Section 9.12.1. The dashed-line rectangle contains an enlarged graphical rendering of the connector permanently attached to the hear. In (a), the solid-line rectangle shows the corresponding variable values from the *end user-device interaction* model. In (b), the solid-line rectangle shows the corresponding variable values from the *affordance* model . 338

9.33 Graphical representation of *end user-device interaction* and *affordance* model final states in the case study counterexample. Letters are added for reference in text of Section 9.12.1. Dashed-line rectangles show enlarged graphical renderings of final states for the old and new controllers. Solid-line rectangle shows corresponding variable values from the end *user-device interaction* model 338

9.34 Graphical representation of *signifier* and *display/control logic* model initial states in the case study counterexample. Letters are added for reference in text of Section 9.12.1. In (a) and (b), rounded-edge rectangles highlighted green indicate what initial states were assigned in the *display/control logic* model 339

9.35 Graphical representation of *signifier* model final states in the case study counterexample. Letters are added for reference in text of Section 9.12.1 340

Abstract

Model-Based Usability Analysis of Safety-Critical Systems: A Formal Methods Framework

Andrew J. Abbate

Amy L. Throckmorton, Ph.D. and Ellen J. Bass, Ph.D.

Complex, safety-critical systems are designed with a broad range of automated and configurable components, and usability problems often emerge for the end user during setup, operation, and troubleshooting procedures. Usability evaluations should consider the entire human-device interface including displays, controls, hardware configurations, and user documentation/procedures. To support the analyst, human factors researchers have developed a set of methods and measures for evaluating human-system interface usability, while formal methods researchers have developed a set of model-based technologies that enable mathematical verification of desired system behaviors. At the intersection of these disciplines, an evolving set of model-based frameworks enable highly automated verification of usability early in the design cycle. Models can be abstracted to enable broad coverage of possible problems, while measures can be formally verified to "prove" that the system is usable. Currently, frameworks cover a subset of the target system and user behaviors that must be modeled to ensure usability: procedures, visual displays, user controls, automation, and possible interactions among them. Similarly, verification methodologies focus on a subset of potential usability problems with respect to modeled interactions.

This work provides an integrated formal methods framework enabling the holistic modeling and verification of safety-critical system usability. Building toward the framework, a set of five, novel approaches extend the capabilities of extant frameworks in different ways. Each approach is demonstrated in a medical device case study to show how the methods can be employed to identify potential usability problems in existing systems. A formal approach to documentation navigation models an end user navigating through a printed or electronic document and verifies page reachability. A formal approach to procedures in documentation models an end user executing steps as written and aids in identifying problems involving what device components are identified in task descriptions,

what system configurations are addressed, and what temporal orderings of procedural steps could be improved. A formal approach to hardware configurability models end-user motor capabilities, relationships among the user and device components in the spatial environment, and opportunities for the user to physically manipulate components. An encoding tool facilitates the modeling process, while a verification methodology aids in ensuring that configurable hardware supports correct end-user actions and prevents incorrect ones. A formal approach to interface understandability models what information is provided to the end user through visual, audible, and haptic sensory channels, including explanations provided in accompanying documentation. An encoding tools facilitates the development of models and specifications, while the verification methodology aids in ensuring that what is displayed on the device is consistent; and, if needed, an explanation of what is displayed is provided in documentation. A formal approach to controlled actuators leverages an existing modeling technique and data collected from other engineering activities to model actuator dynamics mapping to referent data. An encoding tool facilitates model development, and a verification methodology aids in validating the model with respect to source data.

Finally, new methodologies are combined within the integrated framework. A model architecture supports the analyst in representing a broad range of interactions among constituent framework models, and a set of ten specifications is developed to enable holistic usability verification. An implementation of the framework is demonstrated within a case study based on a medical device under development. This application shows how the framework could be utilized early in the design of a safety-critical system, without the need for a fully implemented device or a team of human evaluators.

Chapter 1: Introduction

In safety-critical systems, failures are characterized by a potential "loss of life, significant property damage, or damage to the environment" [16]. In human-interactive systems, usability is the extent to which an interface supports a specified end user in achieving goals effectively and efficiently in a specified operational environment [17, 18]. When considering the potential contributions to successes and to failures with safety critical systems, one should consider the human-system interactions. Safety-critical, human-interactive systems can have complex internal algorithms and controlled actuators, while end users could have varying procedural goals and activities, perceptions and interpretations of the interface, and motor capabilities that shape behavior. Thus, all interacting elements of the human-system interface must be usable. This includes displays and widgets, including their visual, audible, and haptic properties; control systems, including internal algorithms controlling the system's actuators and displays; configurable hardware, such as cables and batteries; and accompanying documentation, such as user manuals and checklists. A broad range of interactions among them must be considered early in the design cycle, and methods and measures are needed to help ensure usability and identify problems that could emerge for the end user.

To support usability of safety-critical systems, standards organizations and regulatory bodies have provided measures that should be tested early in the design cycle [19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30], while human factors and human-computer interaction (HCI) researchers have developed methods for characterizing, designing, and evaluating systems with respect to usability measures [31, 32, 33, 34]. Five broadly categorized measures compiled from the international standardization and regulatory guidance literature are listed below. Text under each standard explains it and provides definitions of italicized items. Examples of complementary design practices/principles are provided, along with examples of their applications in designed systems.

1. The interface should be *accurate* [20, 30, 27, 23, 26, 25]

Table 1.1: Examples of possible end user-interface, controlled actuator-interface, and integrated user-actuator-interface interactions that could emerge in safety-critical systems. Designed elements include interface components and controlled actuators, all of which must be usable in the operational environment. Interactions between documentation and controlled actuators are not considered in this work (denoted by blank cell)

Interface components	End user			Controlled actuators
	Goal-driven activities	Interpretation	Motor capabilities	
Displays and controls	Utilizing input elements to complete procedures	Interpreting symbols, tones, and textures	Physically manipulating switches, knobs, and pushbuttons	Responding to end-user inputs and effecting displayed outputs
Configurable hardware	Configuring components to complete procedures	Interpreting size and shape of cable outputs and connection inputs	Physically manipulating cables and connectors	Connecting to a controller and a power source with cables
Accompanying documentation	Locating procedures and information needed to complete them	Interpreting content	Physically manipulating pages of a printed document	
Possible interactions among above components	Locating and executing procedures involving displays, controls, and configurable hardware in documentation	Interpreting displays, controls, configurable hardware, and content in documentation	Physically manipulating displays, controls, configurable hardware, and pages of a printed document	Responding to end-user inputs, effecting displayed outputs, and connecting to a controller and power source with cables

Accurate refers to correctness of the interface, such as messages rendered on graphical displays that are aligned with particular system states [23, 20], audible alarms that are aligned with particular system failures [27, 35], matching input–output connections of configurable hardware [26, 30], and device descriptions in accompanying documentation that match the functions of components [29, 19].

In support of accuracy, displays and control systems are designed to provide correct information about the system's operational state that evolves due to automation, human inputs, or external stimuli [31]. Configurable hardware is designed for accuracy by making output ends of cables sized and shaped to fit corresponding input sockets; additionally, their design is intended to support configurability for an end user having particular motor capabilities [25] and an environment having spatial constraints [30]. Documentation is designed for accuracy by providing unambiguous descriptions and graphical renderings of system components, such as a labeled, 2-D sketch of a control panel.

2. The interface should be *understandable* [19, 24, 28, 29, 23, 18]

Understandable refers to behavioral characteristics and designed properties of an interface that the end user can interpret with reduced cognitive effort [33]. An understandable interface provides the end user with knowledge of what state the device is in, what actions to take, and what the consequences of actions will be [31]. Control systems are designed for understandability such that actions involving a component or widget will have consistent, expected behaviors with respect to end user knowledge and operational states of the device [36]. Text, tables, and diagrams in documentation are designed for understandability by tailoring the level of technical description to the intended audience and labeling diagrams clearly. Perceivable properties of the device such as graphical display messages, audible alerts, and tactile pushbuttons are designed to leverage recognizable symbols and metaphors; for example, a pushbutton that operates to stop the system could be red, octagon shaped, and slightly concave to fit a fingertip, meaning "push to stop" for end users having knowledge of stop signs and pushbuttons [37].

3. The interface should be *error tolerant* [30, 38, 27]

Error tolerant is a characteristic of an interface that handles erroneous human behaviors in ways that prevent failures. Automated control systems are designed for error tolerance so failures do not occur when end users make common errors, such as omitting procedural steps, repeating them, or performing them out of order [39]. Systems that can respond to inputs from the end user are designed with control logic enabling users to undo the effects of erroneous actions [31]. Configurable hardware components are designed for error tolerance such that erroneous attempts to configure components will not introduce hazardous situations; for example, cable output ends are sized and shaped such that attempts to connect an output end to the wrong input sockets will be unsuccessful.

4. The interface should be *time efficient* [19, 22, 21, 29, 27, 20]

Time efficient is a characteristic of the interface that enables end users to achieve task-related goals expediently. Control systems are designed for time efficiency by updating in real time with the system's operational state; for example, graphical or audible cues engage in a timely manner

when the system's operational state changes, supporting end uses in taking necessary actions quickly [31]. Documentation is designed for time efficiency by incorporating tables of contents, labeled page numbers, and cross-references that support end users in locating procedures and declarative knowledge quickly. In time-critical procedures such as troubleshooting, multi-part steps are decomposed into sets of sub-steps that can be performed quickly; and steps are ordered such that goals can be achieved in a time-efficient way. In this research, time efficiency is defined with respect to the number and temporal ordering of actions, but not the time it takes to complete an action.

5. The interface should be *complete* [23, 29, 27]

Complete is a characteristic of an interface that provides functionality and declarative knowledge enabling end users to set up, operate, and troubleshoot the system effectively [33]. An interface is designed for completeness by using perceivable properties of the device, such as visual symbols, audible tones, and tactile surfaces, to communicate information about system functionality. If symbols, sounds, and textures presented on the device are ambiguous, they must be explained in the content of accompanying documentation. For example, engineering constraints often require interface designers to use coded alerts, such as symbolic shapes and periodic beeps. While it is ideal for coded alerts to be understandable without referencing documentation, potential ambiguities may need to be explained for the interface to be complete.

Control systems are designed for completeness such that sequences of user inputs make progress toward all task-related goals. Procedures in documentation are designed for completeness by addressing possible system configurations; for example, in systems having two possible power sources, a complete procedure provides instructions that are applicable when either power source is in-use.

1.1 Model-Based Verification of Usability

In many engineering domains, model-based design methods are utilized to represent and evaluate complex device prototypes early in the design cycle. Temporally evolving target system behaviors can be modeled and executed in silico, and target systems can be analyzed with respect to desired characteristics using highly automated software technologies [40]. One such set of technologies are

formal methods, originally developed for the modeling, specification, and exhaustive verification of computer systems [41].

At the intersection of human factors, HCI, and formal methods, researchers have developed methods and measures for modeling and evaluating human-interactive systems formally [42]. Models can be abstracted to enable broad coverage of possible problems, and specifications can be encoded and verified to "prove" that the target system is usable. Extant methods and measures support the analyst in specifying and verifying some aspects of accuracy, understandability, error tolerance, time efficiency, or completeness (Fig. 1.1).

1. Specifications related to accuracy:
   (a) Behavioral consistency [43, 44]: A specific action will always result in a specific state of the interface, controlled system, or both
   (b) Weak task connectedness [45]: Starting from any state, there is at least one way to get to a particular goal state
   (c) Strong task connectedness [45]: Starting from any state, the end user can always get to a specific next-state using a particular action
2. Specifications related to understandability:
   (a) Absence of mode confusion [46, 47, 48, 49, 50]: Emergent states of the target system will always match end-user expectations
3. Specifications related to error tolerance:
   (a) Reversibility [45, 43, 51]: The effects of an action can be undone in one action
   (b) Recoverability [43, 52, 51]: The effects of an action can be undone in one or more actions
   (c) Robustness [53, 54, 55]: The interface control logic (including automation [56]) is designed in a way that helps prevent failures in response to erroneous human actions
4. Specifications related to time efficiency:
   (a) Task efficiency [52, 54]: An end user can achieve a task-related goal in a specified number of steps (or fewer)
   (b) Deadlock freedom [45, 57]: The interface will always respond to at least one human input
   (c) Feedback [43, 52, 58]: A specific action will always cause an observable change to the interface
5. Specifications related to completeness:
   (a) Weak task completeness [45, 59]: Starting from an initial state, there is at least one way for the user to complete a procedure
   (b) Task liveness [57]: A human operator can always begin executing at least one task
   (c) Strong task completeness [45, 43]: Starting from any state, the user can always complete a procedure

Figure 1.1: A subset of specifications that have proven useful within safety-critical system analyses, organized as they relate to accuracy (1), understandability (2), error tolerance (3), time efficiency (4), and completeness (5). Numbers and letters serve as a reference for the integrated frameworks in which they have been utilized

Integrated modeling and verification frameworks support the analyst in "proving" these specifications using software-augmented tools and techniques [60, 46, 10, 59, 61, 47, 48]. Theorem proving is a semi-automated technique that involves inductive, interactive generation of specification proofs; in

model checking, specialized software searches a formal model exhaustively for specification violations. Model checking is advantageous because the verification process is fully automated; however, the apparatus must have sufficient computational capabilities for analyzing the model and specification exhaustively [62]. Thus, researchers have focused on developing model checking-based methodologies that are detailed enough to provide useful design insights without overwhelming a model checker (discussed further in Chapter 2).

While extant frameworks vary in scope and level of detail, they generally provide tools and techniques for the formal modeling, specification, and verification of:

- Designed components of the device, including:
  - Displays and controls [59, 61, 47, 10, 46, 59, 48]
  - Controlled actuators [46]
- End-user interaction with the device, which is shaped by:
  - Goals and activities [47, 10, 61]
  - Interpretation of the interface [61]

Some frameworks, such as IVY [59] and ADEPT [48], enable the analyst to model and verify interactions among displays, controls, and human actions that execute independently of a particular task-related goal or operational environment. IVY [59] provides a graphical development environment, including a plug-in [63] for uploading interface source code and deriving formal models using point-and-click tools. Built-in usability specifications and model checking software enables the analyst to automatically verify properties of the interface regarding accuracy (Fig. 1.1, 1a), completeness (Fig. 1.1, 5a, 5c), and error tolerance (Fig. 1.1, 3a, 3b).

The ADEPT framework [48] is a graphical prototyping environment for representing input/output behaviors of displays, controls, and the system's automation. The analyst can model commands that are inputs to the interface executed by the end user, observations that are perceptual and cognitive functions of the end user, and internal actions that are the system's automated control

algorithms [64]. An integrated verification methodology enables the analyst to evaluate understandability of the interface via automated detection of mode confusion [65] (Fig. 1.1, 2a). While the end user's cognitive, perceptual, and motor actions are represented in the model, behaviors are not structured in a goal-driven way.

Other frameworks, such as Shared Event-B [60], the cognitive framework in [61], and the human-automation interaction framework in [46] integrate models of the interface and goal-driven user behaviors. The shared Event-B framework [60] leverages the task modeling notation of Concur Task Trees (CTT) [66] for representing end-user cognitive and motor task behavior. Manually encoded formal models of interface displays and controls are composed with the user model to abstract human-system interaction via the exchange of input/output variables. Human-system interaction is abstracted via the exchange of input/output variables between interface and task models. The framework incorporates a verification methodology enabling analyses of accuracy (Fig. 1.1, 1c, 1b) via theorem proving [67].

In [61], researchers leverage the native syntax of Symbolic Analysis Laboratory (SAL) [68] to integrate a user model representing tasks and abstracted cognitive functions with a device model representing displays and controls of the interface. In the task model, a set of possible motor actions are derived from the user's current goal. In the cognitive function model, an end-user interpretation is derived from the appearance of a visual display message. Updates to what is displayed operate as cues triggering one human action from a set of actions that are possible based on the current goal and interpretation. Specifications assert accuracy (Fig. 1.1, 1b) and completeness (Fig. 1.1, 5a) properties that can be verified automatically using an integrated model checking technique.

In [46], researchers utilize the native syntax of SAL to model the end user, interface displays and controls, and continuous actuator behaviors. The user model represents an internalized conceptualization of the system's internal algorithms that humans are theorized to construct during interaction (called a mental model [36]). This model specifies what the end user believes is the system's current operational state, and it updates when the interface changes states. An interface model represents displays and controls that respond to end-user inputs and automated events that

are trigged by controlled actuator states. A plant model captures approximate actuator behaviors by abstracting them from differential equations (discussed further in Chapter 3, Section 3.2.2). A constraints model, specialized specifications, and a model checking technique ensure that actuator states deemed realistic are considered within automated verification of understandability (Fig. 1.1, 2a).

OFAN [47] and EOFM [10] enable the analyst to integrate models representing conditions in the operational environment within system models representing end-user interaction with displays and controls. The OFAN framework [47] leverages the graphical notation of Statecharts [69] to support the modeling of displays, controls, system automation, the operational environment, and end-user tasks that are goal-driven. Each element within this system model is composed of one or more hierarchical sub-models representing subsystems. Subsystem models interact by responding to updates synchronously or asynchronously over time via the exchange of input/output variables. For example, a model representing displays may respond to updates in a model representing automation synchronously, abstracting the nearly instantaneous travel of electrical signals; while the same display model may respond to updates in a user task model asynchronously, reflecting the relatively slower actualization of human motor actions [70].

In [52], researchers extend the OFAN framework with specifications and model checking capabilities to automatically verify accuracy (Fig. 1.1, 1), completeness (Fig. 1.1, 5), error tolerance (Fig. 1.1, 3b), and time efficiency (Fig. 1.1, 4).

The Enhanced Operator Function Model (EOFM) [10] includes a custom, XML-based [71] grammar for representing goal-driven end-user task behavior as hierarchical-heterarchical activity structures. Decomposition operators specify the temporal and cardinal ordering of activities, which can be decomposed into lower level sub-activities or human actions (actions are at the lowest level and they cannot be decomposed further). The translation tool described in [10] transforms XML-EOFM representations to formal models, and an extension to the translation tool augments normative task models with common human errors [72] such as skipping activities and performing them out of order [56]. Task models can be visualized using the automated macro described in [73], and an integrated

model checking methodology enables automated verification of error tolerance (Fig. 1.1, 3c), time efficiency (Fig. 1.1, 4b), and completeness (Fig. 1.1, 5b) [57].

1.2 Knowledge Gaps

Extant formal methods-based frameworks at the intersection of human factors, HCI, and formal methods have mainly focused on displays, widgets, and control logic as the set of human-system components that shape normative end-user task behavior. Formal verification has proven useful for evaluating some aspects of accuracy, understandability, time efficiency, error tolerance, and completeness. However, few researchers have considered documentation; end-user interpretation of audible, visual, and haptic displays; configurable hardware; and a broad range of interactions among human-interactive system elements. New tools and techniques are needed to support highly automated searches for usability problems in the holistic human-system interface.

While much work has been done in the formal verification of procedures, one set of tasks that has not been addressed involves navigating to documentation pages containing necessary content. Similar to a device's display screens having different symbols and widgets, pages of a printed or electronic document have declarative knowledge such as explanations of what is displayed and what procedures are necessary. For example, if an alarm engages on the device, the end user may need to navigate to a page containing the troubleshooting procedure. If what is displayed on the device is ambiguous, the end user may need to navigate to a page containing declarative knowledge mapping what is displayed to a particular meaning. Navigational tools in accompanying documentation, such as cross-references and tables of contents, should support end users in quickly navigating within and between sections of the document to locate such information. A model-based approach could be developed to ensure documentation navigability and identify potential problems early in the design cycle, similar to the way researchers have addressed navigability between screens of graphical displays [45].

Another aspect of documentation usability involves the way procedures are written. For example:

- If the device components involved in a procedural step are not identified accurately, the end

user may perform the task incorrectly

- If the instructions are not applicable to all possible device configurations, the end user may not be able to complete them

- If procedural steps are not logically ordered, the end user may not execute the procedure in a time-efficient way

Existing modeling methodologies have mostly considered normative end-user task behavior, where the formal task model encoding process is often informed by an existing system's documentation (see for example [74]). Researchers have leveraged the formal task modeling process to aid in identifying potential usability problems with a target system's displays and control logic (see for example [75]); however, such an approach has not been applied toward identifying potential usability problems with written procedures.

End-user inputs to displays and widgets have been modeled and analyzed with respect to a variety of usability-related specifications; however, existing methodologies are limited with respect to configurable hardware. As mentioned, one area concerns interactions with documentation, such as whether instructions address all possible system configurations. Another area concerns interactions between configurable hardware, end-user motor capabilities, and constraints imposed by the spatial environment. For example, the end user must be able to connect cable outputs to corresponding inputs; and if opportunities emerge for the end user to configure hardware incorrectly, failures could occur.

In support of understandability, usability standards identify the need for visual, audible, and haptic properties of the device that are interpretable to the end user with reduced cognitive effort [28, 24]. In support of completeness, accompanying documentation must explain the functions and meanings of potentially ambiguous symbols, sounds, and textures [29, 19]. While the cognitive interpretation framework in [61] incorporates a user model of visual display interpretation, audible properties, haptic properties, and explanations in accompanying documentation are not considered. Additionally, specifications in [61] assert accuracy and completeness with respect to task-related goals; however, specifications of understandability are not provided.

In regard to controlled actuators, one existing framework models actuator dynamics using abstract representations of differential equations [46]. Similar techniques have been utilized successfully in a variety of safety-critical, human-interactive system applications [76, 77, 78, 79, 49] (further detail provided in Chapter 2); however, they require knowledge of differential equations, and formal models are imprecise with respect to actual target system dynamics.

Finally, while each framework discussed in Section 1.1 can incorporate multiple models of the user, interface, operational environment, and controlled actuators, a holistic methodology is needed to ensure usability of the integrated system. For example:

- Alarms must engage on the device when actuators malfunction
- The meanings of alarms must be understandable to the end user
- Troubleshooting instructions must be located quickly in accompanying documentation
- Procedural steps must be accurately written, logically ordered, and applicable to all device configurations
- The end user must be able to configure hardware correctly in the spatial environment

Thus, an integrated approach is needed to model the interactions among controlled actuators, the target system's control logic, device displays, accompanying documentation, configurable hardware, and end-user capabilities. Additionally, specifications and verification methodologies are needed to "prove" accuracy, understandability, error tolerance, time efficiency, and completeness with respect to the holistic, human-integrated system.

1.3 Objectives

To address some of the knowledge gaps in this design space, five objectives of this research address formal modeling, specification, and verification methodologies that are needed to support integrated analyses. Work in these areas should provide useful analytic capabilities that extend existing frameworks while building toward an integrated framework enabling usability analyses with respect to

interacting models of the interface. It would be beneficial for such a framework to incorporate possible interactions among displays, controls, configurable hardware, documentation, the end user, and actuators controlled by the interface. What is needed to address knowledge gaps regarding documentation, configurable hardware, interface interpretation, and controlled actuators are identified and discussed in Sections 1.3.1–1.3.5. The overarching objective of an integrated framework is discussed in Section 1.3.6.

1.3.1 Navigability of Accompanying Documentation

As mentioned, one critical aspect of documentation involves navigability. Navigable documentation is critical for supporting time efficiency of the interface by enabling end users to quickly locate pages containing necessary content. One way designers support time efficiency is by incorporating navigational tools, such as labeled page numbers, cross-references, tables of contents within printed documents, or "help" sections within electronic documents. Examples of printed documents include spiral bound user manuals and paper checklists that are provided with the device; examples of electronic documentations include PDF user manuals or digital checklists that the end user can download from a manufacturer's website.

While it is possible to model navigation tasks using heavily detailed representations (e.g. keystroke-level models [80]), encoding them can be labor intensive. Considering the importance of navigational tools that support time efficiency, analysts could benefit from a formal methods-based approach that could help provide navigability insights using simpler models. A modeling methodology should provide a way of representing the location of content in documentation and the end user's navigation tasks. A verification methodology should support the analyst in ensuring that navigational tools enable the end user to locate content in a time-efficient way.

1.3.2 Usability of Procedures in Documentation

Safety-critical system documentation provides instructions that are necessary for setting up, operating, and troubleshooting the device in its possible configurations. Text and diagrams are often utilized to identify what components are involved in the task, what the task execution conditions

are (e.g., what device malfunction a procedure helps address), and the ordering of procedural steps. Text often prescribes how steps should be executed, while a diagram on the page may be referenced identify the names, appearances, and part-whole compositions of device components. An enumerated list of steps often identifies the normative ordering of tasks, and nested sub-steps often provide individual actions and different sets of tasks that are applicable to particular device configurations [81]. Thus, another aspect of documentation usability involves text, tables, and diagrams that provide accurate, complete, and time-efficient instructional procedures. At a minimum, the procedure should be:

- Be applicable to all system configurations
- Unambiguously describe what component(s) are involved in each task
- Provide logically-ordered steps

A formal methods-based approach could support the analyst in ensuring that such procedures are usable early in the design cycle. A modeling methodology should be capable of representing an end user executing a procedure as-written. It should provide a way of specifying when end-user actions execute (based on the ordering of procedural steps), what end-user actions execute (based on what components are identified in text/diagrams), and what end-user actions are possible (based on configurations addressed in different sets of sub-steps). A verification methodology should support the analyst in verifying accuracy, time efficiency, and completeness with respect to an instantiated model.

1.3.3 End-User Capabilities to Configure Hardware in the Operational Environment

As mentioned, one way hardware is designed for configurability is by ensuring that cable output end and input sockets are appropriately sized and shaped [30, 26]. An accurate design supports the end user in configuring hardware correctly, such as connecting a cable output end to the appropriate input socket, while an error-tolerant design prevents the end user from actualizing erroneous connections. The end user's motor capabilities to move objects must also support hardware configurability. Where

components and the end user are in the spatial environment may also affect abilities to configure hardware correctly.

Considering the need for ensuring that configurable hardware is accurate and error tolerant, extant approaches in formal methods could be extended to enable modeling and verification of such characteristics early in the design cycle. A modeling methodology should enable the analyst to represent the hardware components of a human-interactive system, what configurations are possible in the spatial environment, and what actions the end user can execute when configuring the system. A verification methodology should provide a way of ensuring that hardware components are designed to ensure accuracy and error tolerance for a specified end user and environment.

1.3.4 End-User Interpretation of the Interface

In support of understandability, perceivable properties of interface components such as text on a visual display, volume of an audible alert, and intensity of a haptic vibration are designed to be interpretable with reduced cognitive effort [33]. When incorporating such properties within an interface, designers may consider cultural context, such as a red octagon pushbutton emulating a stop sign to mean “stop.” They may also consider the end user’s perceptual capabilities, such as the ability to perceive and describe identifiable colors, sounds, textures, and vibrations. To support understandability, designers may incorporate properties that operate synchronously through visual, audible, and haptic channels; and what is signified must be consistent within and between properties operating through all sensory channels. To support completeness, what these properties mean may also be explained through the documentation channel, such as text within a user manual describing the consequences of pushing a red octagon button.

Considering the need for an understandable and complete interface, analysts could benefit from formal modeling and verification methodologies extending the capabilities of extant frameworks. A modeling methodology should provide a way of representing what information could be provided to the end user via perceivable properties of the device and explanations in accompanying documentation. It should support the analyst in representing what device components have such properties; which ones operate through visual, audible, and haptic, channels respectively; what they mean to

the end user; and what changes to the properties and their meanings can occur as the system evolves. A verification methodology should provide aid in ensuring that the interface is interpretable to the end user in a way that supports understandability and completeness.

1.3.5 Continuous Actuator Dynamics

As mentioned, further study is needed in the formal modeling of actuators controlled by the interface. The analyst may want to represent precise actuator dynamics, and she may prefer to encode such a representation without the need for differential equations. Thus, a new modeling methodology could be beneficial. Such a methodology should support the analyst in representing the behaviors of actuators controlled by the interface, without the need for differential equations or approximation.

1.3.6 The Integrated Human-System Interface

The objectives discussed thus far have been building toward a formal methods-based framework for ensuring safety-critical, human-interactive system usability early in the design cycle. Such a framework should enable the analyst to model an interface, including documentation, displays, controls, and configurable hardware; an end user, including task behavior, interpretation of the interface, and abilities to configure hardware; an operational environment, including where components are in relation to each other and the end user; and actuators controlled by the interface. The framework should provide a way of abstracting possible interactions among these elements, such as those identified in Table 1.1. A verification methodology is needed to help ensure that the holistic human-system interface is accurate, understandable, error tolerant, time efficient, and complete with respect to modeled interactions.

1.4 Contributions

In support of the objectives identified in Section 1.3, the contributions of this research extend the scope and analytic capabilities of model-based usability verification frameworks (Table 1.2).

Chapter 2 provides a broader conceptual background of human factors, formal methods, and the intersection of these fields, including what tools and techniques are needed to formally model human-interactive systems and verify usability specifications. To support the development of formal models,

Table 1.2: Broad overview of the scope covered by extant modeling and verification methodologies (works referenced in cells), areas addressed in this research (chapters referenced in cells), and areas that should be explored in future work. Interactions between documentation and controlled actuators are not considered in this work

Interface components	End user			Controlled actuators
	Goal-driven activities	Interpretation	Motor capabilities	
Displays and controls	[46, 10, 59, 61, 47, 48]	[61], extended in Chapter 7	Future work	[46], extended in Chapter 8
Configurable hardware	Future work	Future work	Chapter 6	Future work
Accompanying documentation	Chapter 4	Chapter 5	Future work	
Interactions among all of the above	Chapter 9	Chapter 9	Chapter 9	Chapter 9

analysts could benefit from formalisms, modeling techniques, and model development tools. To support formal verification, analysts could benefit from specifications and model checking techniques. Chapter 3 provides a description of the encoding techniques and apparatus employed in this research. Chapters 4–8 introduce new methodologies that extend the capabilities of existing frameworks, building toward Chapter 9: an integrated framework for modeling, specifying, and verifying the holistic human-system interface. Chapter 10 provides a broader discussion of contributions and future work, and model code listings are provided in appendices.

1.4.1 A Formal Approach to Documentation: Modeling, Specification, and Verification of Navigability

The approach developed in Chapter 4 addresses the objective identified in Section 1.3.1. It encompasses a modeling methodology, a verification methodology, and a case study application.

A formalism and modeling technique is developed to represent pages of a printed or electronic document as well as a constrained set of ways the end user could navigate through it: staying on a current page that contains necessary content, navigating one-page forward if there is unfinished content on the current-page, and navigating to a page that is cross-referenced on the current-page.

To employ the methodology the analyst should consider the document's navigational tools (e.g. hyperlinks in a PDF) and the content on each page that could enable one or more possible next-pages (e.g. multiple cross-references).

A set of three specifications regarding time efficiency of documentation navigability and a model checking technique support the analyst in automatically verifying that the end user can navigate to and from necessary pages utilizing the behaviors represented in an instantiated model.

These methodologies are applied in a case study based on a medical device PDF user manual having procedures, information about the device, tables of context, and hyperlinking functions. Case study results indicate that the approach shows promise for enabling verification of time efficiency with respect to a constrained set of end-user behaviors and specifications. Contributions include modeling and verification methodologies that extend the capabilities of extant formal methods-based frameworks with respect to documentation navigability.

1.4.2 A Formal Approach to Documentation: Modeling, Specification, and Verification of Procedure Usability

The approach developed in Chapter 5 addresses the objective identified in Section 1.3.2. It encompasses a modeling methodology, a verification methodology, and a case study application.

Leveraging an existing task analytic framework [10], a modeling technique is developed to represent sets of end-user actions that are possible based what device components are identified in the content of procedural steps. The approach aims to support the analyst in identifying potential, accuracy-related usability problems while attempting to encode the procedure formally, where accuracy problems could emerge from insufficient descriptions of components and parts provided in the instructions.

A device modeling technique provides a way of encoding initial component configurations that are possible when a procedure begins executing vis-a-vis configurations addressed in prescribed instructions. By inspection, if all possible initial configurations are addressed in the procedure, such as either mutually exclusive power source being in-use, the instructions can be considered complete. Otherwise, there may be completeness problems.

Finally, in support of ensuring time efficiency, two generalizable specifications and a model checking technique enable the analyst to verify that procedural steps are logically ordered. Each specification asserts a particular ordering of procedural steps that could be problematic, depending on the procedure's purpose (e.g. setup, troubleshooting). The verification technique leverages two versions of a model representing:

1. The end user executing all procedural steps as-written and in the prescribed order

2. The end user executing one or more procedural steps as-written, but in any order

A specification that is proven in the first model indicates that the procedure has potential time-efficiency problems, while specification violations in either model generate a trace through it reflecting a potentially time-efficient ordering of steps. The modeling, specification, and verification methodologies are demonstrated in a case study based on a medical device troubleshooting procedure.

Case study results indicate that:

- The task modeling methodology could be useful for identifying potential accuracy problems involving what components are identified in task descriptions

- The device modeling methodology could be useful for identifying potential completeness problems regarding possible initial system configurations versus those addressed in the procedure

- The verification methodology could be useful for identifying potential time-efficiency problems as well as revealing potentially improved orderings of procedural steps

Contributions include a novel application of an existing task analytic framework, extending the capabilities of formal methods-based approach with respect to procedures in a system's accompanying documentation.

1.4.3 A Formal Approach to Hardware Configurability: Modeling, Specification, and Verification of Gibsonian Affordance

The approach developed in Chapter 6 addresses the objective identified in Section 1.3.3. It encompasses a modeling methodology, a verification methodology, a tool facilitating the analysis, a case

study application, and a scalability evaluation.

Methodologies are developed to support the modeling, specification, and verification of opportunities for end-user motor actions that emerge in an operational environment (called affordances [82]). To apply the methodology the analyst should consider the part-whole composition of hardware components and the end user's motor capabilities. This information is utilized in conjunction with a modeling technique and encoding tool that facilitates the process of developing a formal affordance model based on one of three affordance formalisms from ecological psychology [6, 4, 7]. A complementary human-environment system (HES) modeling technique supports the analyst in representing initial end-user motor capabilities, initial spatial relations among components and the user, and spatial relations that emerge after the end user takes an available action (called "actualizing" an affordance [6]).

The verification methodology includes four generalizable specifications and a model checking technique. The analyst can verify hardware configurability with respect to an instantiated HES-affordance model composition, two accuracy-related affordance specifications, and two error tolerance-related affordance specifications.

The approach is applied in a case study based on a medical device adverse event involving an incorrect input–output cable connection that occurred during pacemaker implantation surgery [83]. One affordance specified in the model represents spatial relations and motor capabilities of the surgeon enabling incorrect cable connectability, while a second affordance represents spatial relations and motor capabilities enabling a correct one. One accuracy and one error-tolerance specification are instantiated and verified using model checking. A scalability evaluation is also included.

Case study results indicate that the approach could be useful for ensuring accuracy and error tolerance as well as identifying potential usability problems with respect to two cable connectability affordances. Scalability results indicate that the approach shows promise for conducting symbolic model checking in formal models representing up to 32 affordances (and 7.3×10^{134} states) on a desktop workstation having 64 GB of RAM.

Contributions include the modeling methodology, verification methodology, and tool support

extending the capabilities of extant formal methods-based frameworks with respect to hardware configurability.

1.4.4 A Formal Approach to Interface Interpretation: Modeling, Specification, and Verification of Signifiers

The approach developed in Chapter 7 addresses the objective identified in Section 1.3.4. It encompasses a taxonomy modeling methodology, a verification methodology, a tool facilitating analyses, a case study application, and a scalability evaluation.

The modeling and verification methodologies in this chapter are based on the theory of signifiers: clues providing insights into the function, purpose and meaning of the system, component or widget [37]. Theories of perception from psychology and HCI are leveraged to identify a set of perceivable interface properties that can operate as signifiers in different ways [6, 84, 85, 86]. To instantiate a formal signifier model, the analyst should consider the end user's perceptual capabilities and contextual factors (e.g. culture, knowledge of the system) that could control what is signified by perceivable properties of the device and explanations in accompanying documentation. Such a model represents:

- Functions and meanings that could be signified on the device and explained within accompanying documentation (e.g. "low battery" for the meaning of an indicator light)
- Device components (e.g. an indicator light)
- A constrained set of perceivable properties of device components that operate through visual (e.g. color), audible (e.g. volume), or haptic (e.g. vibration) channels respectively
- What identities the end user could assign to perceivable properties (e.g. "red" for a color)
- What function or meaning is signified to the end user based on her description of the property (e.g. "system stopped")
- What is signified through explanations in the content of documentation (e.g. text explaining what is meant by an indicator light that the end user describes as colored "red")

- What changes to end-user descriptions of perceivable properties and what is signified through audible, visual, haptic, and documentation channels as the system evolves

An encoding tool facilitates model development. The verification methodology provides two generalizable understandability-related specifications, one generalizable completeness-related specification, and a model checking technique. Complementary device and user modeling techniques support the analyst in representing changes to displayed perceivable properties that emerge due to end-user inputs and the system's own algorithms. The approach is demonstrated in a case study based on a medical device and its interface, including accompanying documentation. A scalability evaluation is also included.

Case study results indicate that the approach shows promise for ensuring understandability and completeness as well as identifying potential usability problems involving conflicting signified information on the device and in accompanying documentation. Scalability results indicate that the approach shows promise for modeling up to 64 signified functions or meanings for a device having up to 128 modes and verification via symbolic checking on a desktop workstation having 64 GB of RAM.

Contributions include the signifier taxonomy, modeling methodology, verification methodology, and tool support extending the capabilities of extant formal methods-based frameworks with respect to end-user interpretation of an interface.

1.4.5 A Formal Approach to Controlled Actuators: Modeling of Continuous Device Dynamics Derived from Spreadsheet Data

The approach developed in Chapter 8 addresses the objective identified in Section 1.3.5. It encompasses a modeling methodology, a tool facilitating formal model development, a case study application, and a scalability evaluation.

Leveraging a technique from formal methods (discussed further in Chapter 3, Section 3.2.2), the methodology facilitates formal modeling of actuators controlled by the interface, where the model generates outputs derived from data stored in one or more tabulated spreadsheets instead of differential equations. A formal device modeling technique supports the analyst in representing actuator

dynamics as an input/output function. Alone, the function has an infinite range of continuous input/output parameters. To address this, function parameters are constrained using additional model infrastructure derived from data collected beforehand via other engineering activities, such as computational fluid dynamics [87]. An automated tool generates this infrastructure, and an accompanying verification methodology ensure that inputs and outputs of the formal model match those represented in spreadsheet data. The approach is demonstrated in a case study using data from a medical device under development. A scalability evaluation is also included.

Case study results indicate that approach is capable of modeling actuators controlled by the interface in a way that is correct with respect to spreadsheet data. Scalability results indicate that the approach shows promise for representing actuator behaviors derived from a spreadsheet having up to 65,536 cells. Contributions include a new way of modeling continuous device dynamics, without the need for differential equations or approximation.

1.5 An Integrated Framework for Verifying Safety-Critical, Human-Interactive System Usability

Chapter 9 addresses the objective identified in Section 1.3.6, the overarching aim of this research: an integrated framework enabling the formal modeling, specification, and verification of safety-critical, human-interactive systems. In support of this objective, Chapter 9 encompasses a modeling methodology, a verification methodology, and a case study application.

In extant frameworks discussed in Section 1.1, the underlying theory is that a human-interactive system is an integrated composition of normative end-user goals/activities and interface displays/-controls. The theory underlying this framework treats the human-interactive system as an integrated composition of:

- Documentation, including:
  - Navigational tools
  - Instructional procedures
  - Device information

- A device, including its:
  - Displays and controls
  - Configurable hardware
  - Controlled actuators
- An end user, including three factors that shape behavior:
  - Perceptual capabilities
  - Motor capabilities
  - Contextual factors such as cultural background and knowledge of the system
- A spatial environment containing documentation, the device, and the end user

To employ the framework (Fig. 1.2), the analyst considers how system elements could interact and encodes a formal model representing such interactions. In one potential implementation, each element of the framework is encoded using methodologies developed in earlier chapters. A modular model architecture and augmented modeling techniques enable the analyst to abstract a broad range of theorized interactions.

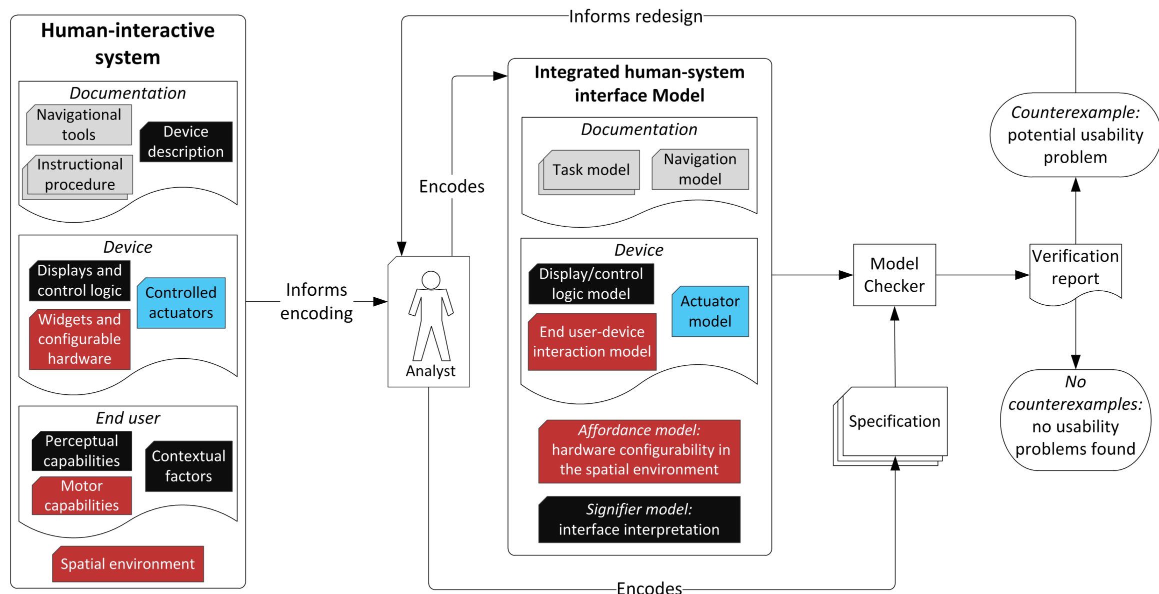


Figure 1.2: Graphical representation of the integrated framework. Elements of the target system are color-coded to identify corresponding framework models

The exchange of input/output variables between framework models reflects constraints on end-user behaviors. Examples (discussed further in Chapter 9) include:

- An end user must navigate to a page listing procedural steps to execute motor actions in the prescribed order
- An end-user motor action effects correct changes to configurable hardware only if the correct affordance exists when the action is attempted
- When a controlled actuator malfunctions, the malfunction is only signified through the documentation channel if:
  - Audible and visual alarms are engaged on the interface,
  - the meanings of both alarms are explained in documentation, and
  - the end user has navigated to the referent page

To enable verification, a set of specifications is developed to assert desired properties of the interface with respect to usability measures involving theorized interactions (Table 1.3). A model checking technique supports the analyst in verifying specifications.

The framework is demonstrated in a case study based on a prototype medical device under development. Formal models represent a draft of printed documentation, an operational and troubleshooting procedure therein, alarms and setting of the device, what is signified on the device and through accompanying documentation, and controlled actuator behaviors. Models are composed using modeling techniques developed to support the integrated architecture. Using this model, each specification in Table 1.3 is verified using infinite-bounded model checking [68].

Case study results indicate that the integrated framework methodologies show promise for ensuring usability and uncovering potential problems early in the design cycle of a safety-critical, human-interactive system. Contributions include a integrated theory of human-system interaction, a formal model architecture representing it, and a verification methodology encompassing a broad range of theorized interactions with respect to intersecting usability measures.

Table 1.3: Descriptions of temporal logic specifications that can be encoded in LTL for an integrated framework model

	Accuracy	*Understand-ability*	*Error tolerance*	*Time efficiency*
Understand-ability	Signifiers are accurate, consistent, and (optionally) redundant while a procedure is executing and the system is in a particular state			
Error tolerance	An unsafe affordance does not emerge while a procedure is executing and the system is in a particular state	An unsafe affordance does not emerge and signifiers are consistent and (optionally) redundant		
Time efficiency	If a procedure is executing and the next-state of the device is different from the current-state, a desired affordance emerges in the next-state	If the next-state of the system is different from the current-state, signifiers are accurate, consistent, and (optionally) redundant in the next-state	If the next-state of the system is different from the current-state, an unsafe affordance does not emerge in the next-state	
Completeness	Signifiers are accurate and complete while a procedure is executing and the device is in a particular state	Signifiers are consistent, (optionally) redundant, and complete	An unsafe affordance does not emerge and signifiers are complete	If the next-state of the system is different from the current-state, signifiers are complete in the next-state

Chapter 2: Conceptual Background

Chapter 1 identified the many interacting components of a human-system interface, how their design is informed in safety-critical systems, and what aspects of human-system interaction can be modeled and verified using formal methods. Considering the interacting elements of a human-system interface with respect to usability standards, this work focuses on extending extant methodologies, providing analyses of the human-interactive system that build toward a holistic, integrated framework. Such a framework should enable early evaluations of human-system interface usability that consider a broad range of interactions among the end user, device, and operational environment. Human factors, safety-critical systems, and formal methods are discussed next to provide background.

2.1 Human Factors and Safety-Critical Systems

To inform the design of usable safety-critical systems, researchers have developed methods to aid in characterizing the user, device, and environmental elements that shape human-interactive system behavior. Human elements include goals and activities, sensory and psycho-motor capabilities, knowledge, and experience; device elements include displays, power sources, configurable hardware, control systems having various automated behaviors, and documentation accompanying the device; environmental elements include lighting, noise, and physical constraints (e.g. size of a room, other objects/agents in the vicinity) [88, 89, 82, 90]. Task analysis concerns end user goals and activities that are involved in setup, operation, and troubleshooting procedures [91]. For example, cognitive task analysis includes task work that is supported by knowledge [92], including:

- Declarative knowledge regarding what objects and actions are involved in tasks,
- Procedural knowledge regarding how to complete tasks, and
- Strategic knowledge regarding the context in which tasks should be executed [93].

Outputs of these analyses are useful for informing the development of interface prototypes; early

in the design cycle, they are useful for evaluating prototypes using a variety of model-based methodologies [40, 42].

2.1.1 Model-Based Design

To ensure that designed systems are accurate, understandable, error tolerant, time efficient, and complete, the analyst must have methods and measures to identify potential problems, test potential improvements, and produce conclusive results. Model-based design methodologies provide one such set of methods and measures.

In safety-critical systems, the model-based design process enables the analyst to encode computational representations of temporally evolving human-interactive system behaviors, and specialized software can search models for usability violations or can use the models to simulate emergent behaviors and predict performance characteristics [94, 95, 35, 20]. A broad range of modeling techniques and evaluation methodologies enable such analyses.

Extant formal modeling techniques range in scope and level of detail with respect to goal-driven task behavior and psycho-motor processes [42, 96, 97, 98, 49]. The majority involve encoding task models that are structured, temporally evolving representations of normative human behaviors that are goal-driven [99]. Cognitive and perceptual elements that are modular or task-integrated can incorporate knowledge of the device, psycho-motor capabilities, and constraints imposed by the operational environment [96, 56, 61].

Human-system interaction can be represented by encoding and composing models of the device and end user in a modular way. Early in the design cycle, models of the device are often based on an operational concept of the proposed system (see [100]) or existing systems that are similar (see [101]). Parametric equations can be encoded to represent the behaviors of displays, controls, and continuous actuators that evolve over time, and human-system interaction can be abstracted via the exchange of input/output variables with a user model [102, 46, 70, 49]. Verification methodologies aid in searching models for problems that could emerge for end users. In this design space, verification methodologies are preferred, since we are interested in detecting, identifying, and correcting usability problems early in the design cycle. Formal methods are one such set of methodologies.

2.2 Formal Methods

Formal methods are a set of well-defined mathematical formalisms, techniques, and technologies that enable model-based verification of desired target system properties (called specifications) [41]. A formalism is a set of symbolic variables whose semantics and mathematical relationships represent behaviors of a particular class of target systems, such as digital computers. Modeling techniques are processes for instantiating a formalism to represent the formal model of one particular system, such as an airplane cockpit having automation and human-input controls [103].

Finite state machines (FSMs) are one such set of formalisms [104], and they have proven useful in model-based design of safety-critical systems [42]. FSMs are made up of symbolic variables representing target system behaviors as a finite set of states, transitions, and next-states [62]. Each state is a unique set of valued variables representing one mutually exclusive configuration of the target system (sometimes called a "mode" [105]), and each transition to a next-state represents forward temporal progress in an abstract way. Transitions can have guards that are Boolean (true/false) expressions that enable a next-state when valued *true*. Many possible transitions can be enabled in a state, and only one can execute during verification. The number of states in a model is its size (or "state space"), and each unique sequence of next-states through the model is called a path. Generally, state space increases with breadth (i.e. number of paths through a model) and depth (i.e. lengths of paths through a model) [106].

Specifications are propositional formulas that assert desired system behaviors using valued variables of an instantiated formal model and the syntax of a temporal logic [62]. The propositional element of a temporal logic specification is made up of valued variables, Boolean operators, logical connectives, and (optionally) variable quantifiers (Table 2.1).

The two temporal logics commonly utilized in formal methods, computational tree logic (CTL) and linear temporal logic (LTL) [62]. Both incorporate the logical connectives and Boolean operators of Table 2.1 as well as the temporal operators shown in Table 2.2. CTL is a branching-time logic that can be used for models in a tree-like structure in which there are different paths in the future, any one of which might be an actual path that is realized. In CTL, path quantifiers (last two rows

Table 2.1: Boolean operators, logical connectives, and example propositions that can be incorporated within temporal logic specifications. ϕ and ψ are hypothetical formal model variables that can be valued 0 or 1

Formula element	Symbol	Name	Example	Interpretation
Logical connective	$=$	Equals	$\phi = 1$	ϕ equals 1.
	$\neq$	Not equal	$\phi \neq 1$	ϕ does not equal 1.
	$\Rightarrow$	Implies	$\phi = 1 \Rightarrow \psi = 0$	ϕ equals 1 implies ψ equals 0.
Boolean operator	$\wedge$	And	$\phi = 1 \wedge \psi = 0$	ϕ equals 1 and ψ equals 0.
	$\vee$	Or	$\phi = 1 \vee \psi = 0$	ϕ equals 1 or ψ equals 0.
	$\neg$	Not	$\neg(\phi = 0)$	ϕ does not equal 0.
Variable quantifier	$\forall$	For all	$\forall \phi(\psi = 0)$	ψ equals 0 for all possible values of ϕ.
	$\exists$	Exists	$\exists \phi(\psi = 0)$	ψ equals 0 for at least one value of ϕ.

of Table 2.2) can be placed in front of temporal operators (first two row of table 2.2) to specify whether the proposition should hold along at least one path or along all paths in a formal model. The CTL "E" quantifier is useful if the analyst wants to know if a state can be reached along one or more paths through the model. Specifications that incorporate a "G" temporal operator are useful if the analyst wants to ensure that an undesired state never occurs along paths through a model; specifications that incorporate an "F" temporal operator for analyses that concern progress toward a desired state; and specifications that incorporate an "X" temporal operator are useful for analyses concerning immediate next-states that are desired [107].

In LTL, one specification invokes all paths, and path quantifiers are not part of its syntax. In computer science, LTL specifications that only incorporate a "G" temporal operator are called "safety specifications," which are useful if the analyst wants to ensure that a safety-critical target system model never enters an undesired state.

Model checking is a highly automated approach that uses specialized algorithms to search formal models exhaustively for temporal logic specification violations. The analyst passes a formal model and a temporal logic specification to model checking software, which searches the model exhaustively for specification violations. If a violation exists, a verification report returns a trace through the

Table 2.2: Propositions from Table 2.1 augmented with temporal operators and path quantifiers that are needed in temporal logic specifications. LTL specifications have temporal operators. CTL specifications have path quantifiers and temporal operators

Formula element	Symbol	Name	Example	Interpretation
Temporal operator	G	**G**lobally	$\mathsf{G}(\phi = 1)$	ϕ always equals 1.
	X	Ne**X**t	$\mathsf{X}(\phi = 0)$	ϕ equal 0 in the next-state.
	F	**F**uture	$\mathsf{F}(\psi = 1)$	ψ eventually equals 1.
Path quantifier	A	**A**ll	$\mathsf{AG}(\phi = 1)$	ϕ always equals 1 along all paths through the model.
	E	**E**xists	$\mathsf{EF}(\psi = 0)$	ψ eventually equals 0 along at least one path through the model.

model leading up to the problematic state (called a counterexample), which could elucidate a potential design problem. If no violations exist, no counterexample is returned, and the target system can be considered safe with respect to the model and specification. Verification can fail if the apparatus has insufficient computational capabilities with respect to the model and specification, illustrating a case where the model must be abstracted to reduce the level of detail therein. Abstraction must be performed carefully to ensure that counterexamples do not represent implausible or impossible behaviors of the target system. If such a counterexample is returned, the model may need to be refined or constrained in a way that affords computable verifications for the apparatus and meaningful counterexamples for the analyst.

2.3 Formal Methods in Human-Interactive Systems

Researchers at the intersection of formal methods and human factors have developed an evolving set of tools and techniques that enable "proving" usability of human-interactive systems early in the design cycle [56, 46, 70, 108, 75]. The goal is for the modeling and verification methodologies to integrate human factors techniques with the formalisms, formal modeling techniques, propositional logic specifications, and verification technologies that are utilized in formal methods, while reducing the need for a multidisciplinary team of experts.

With respect to model-based design, researchers have demonstrated that a formal methods-based

approach can be useful in many ways. Reasoning about human-interactive system design during the model encoding process can aid the analyst in identifying potential usability problems while attempting to represent them formally (see [101] and [108] for examples). Using formal verification via model checking, researchers have analyzed a variety of complex, safety-critical system case studies and interpreted verification reports to identify usability problems (see [75] and [46] for examples).

One area involves models of interface displays and controls (with and without automation) and normative human task behavior [42]. Formalisms, modeling techniques, and tool support aid the analyst in developing modular formal models of the human and device. Human interaction with displays and controls is abstracted by composing modular formal models (discussed further in Section 2.4.2.4). To support formal verification researchers have developed a taxonomy of generalizable, temporal logic specifications, as well as verification techniques and tools [45, 109, 110, 111, 61, 108, 112].

The following sections provide a review of the state of the art, organized in terms of extant tools and techniques (listed in outline form below). The scope of each tool or technique is discussed in each section listed in parentheses, organized in terms of the human-interactive system elements that can be modeled, specified, and analyzed.

- Formal modeling methods that support the analyst in representing the behaviors of displays, control systems, continuous actuators, end users, and constraints imposed by the operational environment; where methods encompass:
  - Formalisms that are needed to identify the mathematical semantics and relationships that represent human-interactive system elements (discussed in Section 2.4.1)
  - Modeling techniques that are needed to develop formal models that are instantiated formalisms (discussed in Section 2.4.2)
  - Tools that facilitate the formal model development process (discussed in Section 2.4.3)
- Verification methods that support the analyst in identifying problems involving interactions among device, user, and environmental elements; where methods encompass:
  - Temporal logic specifications that are needed to represent usability-related properties of

a particular target system (discussed in Section 2.5.1)

– Model checking techniques that are needed to conduct highly automated formal verification (discussed in Section 2.5.2)

2.4 Modeling Methodologies

Researchers have approached formal modeling of human-interactive systems using different combinations of formalisms, modeling techniques, and encoding tools. In any approach, an underlying formalism is needed to define the semantics and mathematical relationships among a set of symbols or keywords representing a class of target systems. Modeling techniques support the analyst in instantiating a formalism to represent a particular system in a formal model that can be analyzed via model checking. Encoding tools integrate formalisms and modeling techniques to reduce the need for manually encoded model checking syntax (see [10]). Formalisms, modeling techniques, and encoding tools are discussed in the following sections to aid in identifying what human-system interaction behaviors can be modeled.

2.4.1 Formalisms

Researchers have developed a variety of FSM-based formalisms for characterizing the behaviors of human-interactive devices, end users, aspects of the operational environment, and the interactions among them that evolve over time [104, 69, 113, 114, 115, 116]. As discussed in Section 2.2, FSM-based formalisms abstract evolving system behaviors as a finite set of states, transitions, and next-states. Symbolic variables such as "S" meaning "the set of states that are mutually exclusive sets of valued variables" and "$\Rightarrow$" meaning "the set of transitions mapping states to next-states" are generalizable with respect to a class of target systems that could include one or more:

1. Devices having discrete digital control logic and configurable hardware
2. Devices having continuously dynamical actuators
3. Humans having cognitive, motor, and perceptual actions

4. Operational environments having objects and humans embedded in environmental conditions such as ambient noise and lighting

The first kind of formalisms are constrained to discrete representations of a device, without representing the end user (where "discrete" refers to symbols representing countable sets of valued variables, such as a list of display messages [61]). Leveraging differential calculus, hybrid formalisms characterize continuous actuator dynamics by augmenting discrete device formalisms with one or more symbols representing infinitely large sets of continuously valued variables [117, 118, 114, 119, 120]. These semantics enable the analyst to characterize systems having both continuous and discrete behaviors, such as aircraft [49], ground transportation vehicles [79], and medical devices [121].

Discrete and hybrid formalisms can incorporate symbols representing user inputs; however, humans are often represented using task analytic formalisms [122, 123, 10, 58, 61]. Extant task analytic formalisms vary in scope, however they commonly incorporate keywords or symbols for representing human states that can be activities, goals, and knowledge. Transition symbols are rule-based functions mapping states to next-states that emerge in response to human actions that can be motor, cognitive, or perceptual [96, 124]. Many such formalisms combine discrete device elements to characterize human-interactive systems that evolve due to human inputs to the device or automated control logic [125, 61, 126].

Formalisms representing the operational environment utilize symbols to represent objects, humans, and the relationships among them, such spatial relations (discussed further in Chapter 6) [8, 3, 2, 9, 115, 1, 7, 4, 6]. They often combine elements of task analytic and discrete device formalisms, where different symbols can represent the end user, device, and operational environment altogether (see for example [115]). Transitions map human, device, and environmental states to next-states that are triggered by human actions, control logic of the device, or stimuli from the operational environment (see for example [70]).

Considering the many interacting elements of a human-system interface discussed earlier, discrete, hybrid, and task analytic formalisms are all needed to analyze safety-critical, human-interactive systems. Task analytic formalisms are needed to model end users; discrete device formalism are needed

to model a human-interactive system's displays, control logic, and hardware configurations; environmental formalisms are needed to model conditions in the operational operational environment; and hybrid formalisms are needed to incorporate continuous actuator behaviors within system models.

2.4.2 Modeling Techniques

In support of formal methods-based analyses, the formalisms discussed in Section 2.4.1 aid in characterizing the many elements that make up human-interactive systems; however they must be instantiated to model one particular target system in this class. Researchers have developed an evolving set of modeling techniques for this purpose. While extant techniques vary in methodology, they can be broadly categorized with respect to what formalism elements can be instantiated:

- Discrete device control logic
- Continuous actuator dynamics
- Human task behavior
- Constraints imposed by the operational environment
- Human-system interaction via the exchange of input/output variables from models representing one or more of the above elements

Discrete device modeling techniques provide ways of instantiating discrete device formalisms as well as discrete states of hybrid formalisms to represent digital control logic of displays and control systems [46, 70, 119]. Continuous device modeling techniques provide ways of instantiating the continuous states of hybrid formalisms to represent actuator dynamics [121, 49]. Task modeling techniques produce normative human behavioral models that, where behaviors are goal-driven and action execution could depend on states of the device, knowledge, and conditions in the operational environment [10, 127, 99, 96, 75].

Combining elements of these techniques, researchers have developed ways of abstracting human-system interaction via the exchange of input/output variables of modular models (see for example

[61]). Graphical modeling techniques provide visually instantiated FSMs, while typesetting techniques leverage the native syntax of a model checking system [46]. General ways of employing them are discussed in the following sections to aid in identifying what human-interactive system behaviors can be modeled in the current paradigm.

2.4.2.1 Discrete Device Modeling Techniques

Formal methods researchers have developed formal modeling techniques that support the analyst in instantiating discrete device formalisms using a graphical notation or the syntax of a particular model checking system [69, 61]. In either approach, states are instantiated to capture the modes of a particular device, while transitions are encoded to represent forward temporal progress in an abstract way. For example, the graphical FSM in Fig. 2.1 is an instantiated discrete device formalism representing the control logic of a household range. A burner light can be on or off, and a numerically labeled, rotatable dial can be adjusted by the end user to a setting between 0 and 4. Initial states are identified to abstract a starting point in time, while transitions to next-states capture how the device responds to an event in the immediate future.

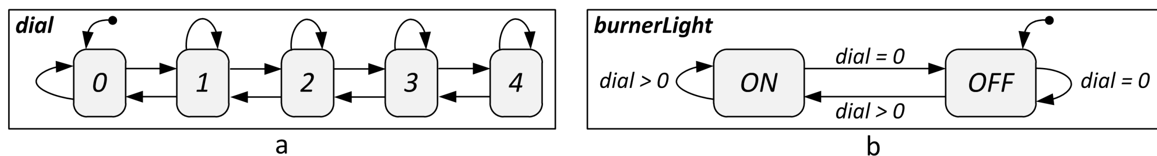


Figure 2.1: Graphical FSM representation of a household range. Variables are boldfaced and italicized within square-edge rectangles. State values are italicized within rounded-edge rectangles. Initial states are indicated by curved arrows having no label and a filled circle. Guarded transitions are indicated by labeled arrows. Unguarded transitions are indicated by unlabeled arrows. (a) The dial having an initial state of 0. It can remain zero or transition to 1. Subsequent states can remain unchanged, transition up-one, or transition down-one. (b) The burner light having an initial state of *OFF*. It remains *OFF* if the dial is set to 0, and it transitions to *ON* if the dial is positive. When the indicator light is in the *ON* state, it remains *ON* if the dial is positive, and it transitions to *OFF* if the dial is set to 0.

To enable formal verification analyses, typesetting techniques commonly employ the syntax of a particular modeling checking system, such as Symbolic Analysis Laboratory (SAL) [68], which has specialized character sequences for representing states, transitions, and next-states. For example, in the SAL code fragment below representing the FSM in Fig. 2.1b, `[]` denotes the start of a guard expression; `-->` denotes the end of a guard expression; and `'` denotes a next-state variable value,

where next-state is denoted by an apostrophe ('). The variable named `burnerLight` transitions to `OFF` when `dial` equals `0`, and to *ON* when dial equals a value greater than `0`.

```
[]dial = 0 -->
  burnerLight' = OFF;
[]dial > 0 -->
  burnerLight' = ON;
```

2.4.2.2 Continuous Device Modeling Techniques

Continuous device modeling techniques enable the analyst to instantiate continuous states of a hybrid formalism, where evolving actuator dynamics are modeled by differential equations [128, 118, 119]. A differential equation represents continuously evolving changes to one or more dependent variables (e.g. actuator speed, electrical current) with respect to one or more independent variables (e.g. time, space). Because digital computing is limited to discrete combinations of 0s and 1s, formal models of continuous system elements rely on approximating differential equation solutions using discrete representations (see [129]). There are currently two broadly characterized ways of modeling continuous system elements in this way:

1. Reachability analysis [128]

2. Relational abstraction [118]

Reachability analysis refers to a variety of approaches in mathematics [130], but in the context of formal methods, it is a technique for approximating differential equation solutions as an abstracted, finite set of possible solutions. They are generally computed by taking a current dependent variable value, calculating a set of future values that are possible (or reachable) in one discrete step, and enclosing them in an ellipsoidal or trapezoidal region of a two-dimensional plot [128]. Reachability analysis has been applied in variety of safety-critical system applications, including aircraft [76, 77, 49], medical devices [78], and motor vehicles [79].

Relational abstraction is a technique that involves encoding the arithmetic relationships between current and future ranges of differential equation solutions [118]. Based on a differential equation, the analyst can encode a relational abstraction using propositional assertions; e.g., a positively

sloped equation must have a future solution that is greater than the current one [46]. Propositions are evaluated as Boolean (true/false) expressions, where a future solution is considered possible if the proposition is valued *false*. Using a linear algebra-based variant of relational abstraction, the analyst can represent systems of differential equations comparing current- and next-state values of an eigenvector multiplied by a dependent variable vector [118]. Multiplying two vectors in this way generates a number that can be incorporated within equality/inequality expressions similar to those utilized in [46].

Like discrete device modeling techniques, the continuous elements of hybrid formalisms must be encoded in a model checking syntax to enable formal verification analyses. An example is provided in Chapter 3.

2.4.2.3 Task Modeling Techniques

A variety of formal task modeling techniques have been developed to support the analyst in representing normative human task behavior by instantiating a task analytic formalism [115, 10, 61, 127]. Normative human task behavior refers to an end user's activities that are performed to achieve goals effectively [131]. Activities and goals can be structured in a hierarchical way such that sub-activities and atomic actions (which cannot be decomposed further) have a temporal and cardinal ordering [10, 66, 132]. Many such techniques provide algorithmic rules for instantiating states, next-states, transitions, and guard expressions that control when activities and actions execute [123, 122].

Task models can be encoded to represent goals and actions that are heterarchical using valued variables and guarded transitions [61]. For example, in the SAL syntax encoded below, an end user's goal is to increase the setting of the household range modeled in Section 2.4.2.1. The guarded transition specifies that when the variable `goal` is equal to `increaseHeat`, the user's next-action is to rotate the dial counterclockwise (denoted by `action' = rotateDialCounterClockwise`).

```
[]goal = increaseHeat -->
  action' = rotateDialCounterClockwise;
```

Task models can also be encoded in a hierarchical-heterarchical way to represent high-level activities, sub-activities, and actions, typically employing the semantics of an accompanying formalism

(discussed in further in Appendix B). For example, the Enhanced Operator Function Model (EOFM) [10] provides a formalism and modeling technique for representing normative human task behavior as a structure set of activities, actions, and execution conditions. Decomposition operators specify the order in which activities/actions can execute (i.e., temporal ordering) as well what activities/actions can execute in relation to each other (i.e., cardinal ordering). Variables representing the device, operational environment, and human cognitive functions are encoded to specify when activities begin, repeat, and complete executing. A graphical notation accompanies the technique to support the analyst in developing visual representations of typeset task models [73] (discussed further in Chapter 5).

2.4.2.4 Human-System Interaction Encoding Techniques

Utilizing modular models of the device and end user, researchers have developed ways of abstracting human-system interaction withing system model compositions via the exchange of input/output variables [42]. While researchers have approached this in many ways [10, 98, 70, 75, 133, 134], a common technique involves composing one formal model of the device with one formal task model. A device model often has output variables representing its operational states that update due to human actions or automation. Outputs operate as inputs to a formal task model, whose outputs are human action variables operating as inputs to the device. This technique enables the analyst to abstract the effects of human actions on the device after a temporal delay, where task and device models update asynchronously (i.e. the task model transitions first, followed by the device model, or vice versa) [10]. For example, in the SAL syntax encoded below, the variable action from the task model encoded in Section 2.4.2.3 operates as an input to the device model encoded in Section 2.4.2.1. The guarded transition specifies that when the user's current action is rotating the dial counterclockwise and the dial's setting is below 4 (its maximum), the dial's next state is increased 1. The current action operating to change a next-state of the device abstracts a temporal delay.

```
[]action = rotateDialCounterClockwise AND dial < 4 -->
  dial' = dial + 1;
```

2.4.3 Tool Support for Developing Formal Models

Tool support facilitates the formal model encoding process in many ways. One set of tools that has proven useful for instantiating formalisms in a model checking syntax leverages formal description languages that are typeset or graphically encoded. Formal description languages usually provide a formalism as a set of keywords and a modeling technique as a hierarchical-heterarchical structure that is required for representing human task behavior or human-system interaction in a static way [132, 66, 73, 134, 135]. Encoding environments support the analyst in applying the technique correctly using a variety of features such as syntax checking and autocomplete [59, 136, 137, 138, 139]. Visualization macros have also been developed to convert typeset representations to sets of shapes and arrows that are treelike structures or graphical FSMs [10, 140, 63]. Instantiated representations are not amenable to formal verification; however, researchers have developed custom, automated translators that can parse formal descriptions and automatically generate a representation that can be verified using model checking [141, 138, 10, 142, 99, 143]. Further details on extant tool support are provided in Appendix B.

2.5 Verification Methodologies

Researchers have approached formal verification of human-interactive system models using different combinations of temporal logic specifications and software facilitating the analysis [59, 48, 144, 140, 46, 57]. As mentioned, a temporal logic specification is needed to encode desired characteristics of the human, device, operational environment, or composable human-system interaction model. Model checking techniques support the analyst in selecting an appropriate model checker for the model and specification. Tool support aids in automatically generating specifications based on a formal model and an underlying, usability-related standards. Specifications and model checking techniques are discussed next to aid in identifying what analyses are currently enabled.

2.5.1 Specifications

To enable formal verification via model checking, the analyst needs to encode a specification using variables encoded in a formal model and temporal logic syntax (Tables 2.1 and 2.2). To support

the analyst, researchers have developed an evolving taxonomy of generalizable specifications based on usability-related design principles to assert desired characteristics of human-interactive system elements. Considering usability-related standards for a safety-critical system interface, one way to categorize extant specifications involves their scope with respect to understandability, accuracy, time efficiency, error tolerance, and completeness. A subset of specifications that have proven useful within safety-critical system analyses are listed and described in outline form below (as in Chapter 1).

To enable formal verification, these specifications are commonly encoded in the syntax of a temporal logic such as CTL or LTL [62]. They can be encoded manually for a particular analysis using generalizable patterns (as mentioned above) or automatically generated using custom tools [57, 59]. To support formal verification of temporal logic specifications, researchers have developed model checking techniques that are applicable to different analyses [46, 145, 59, 10, 48, 146].

2.5.2 Model Checking Techniques

In this work, a model checking technique primarily involves selecting a software tool that is appropriate for a particular analysis. Applicability generally depends on the verification apparatus, the instantiated formalisms (i.e. discrete device, hybrid, or task analytic formalisms), and the specification [68]; thus, researchers have focused on providing a model checking technique as part of an integrated formal methods-based approach [46, 10, 48, 59, 147]. Two broadly categorized techniques that have been employed successfully in safety-critical, human-interactive systems include symbolic model checking [148] and bounded model checking [149].

Symbolic model checking techniques have proven useful for verifying LTL or CTL specifications in human-interactive system models based on discrete device and task analytic formalisms. Specialized algorithms construct a symbolic representation of the state space [150], and the model checker searches the state space exhaustively for specification violations [148]. Executing symbolic model checking successfully requires the analyst's computer to have sufficient memory for storing the symbolic state space representation [151].

Bounded model checking techniques utilize a different set of algorithms [149] that do not involve constructing a symbolic representation of the entire state space. They have proven useful for

verifying LTL specifications in human-interactive system models based on hybrid formalisms and aspects of task analytic formalisms, such as user knowledge of what operational state the device is in [46, 145]. In comparison with symbolic model checking, bounded techniques show promise for conducting systematic analyses involving a broad range of human-interactive system elements, including actuators having continuously evolving states.

Chapter 3: Methodologies and Apparatus

Modeling and verification methodologies are used to develop formalisms and formal models in the remainder of this document. Different kinds of specifications and model checking tools are applied toward different analyses. For reference, they are described in this chapter.

3.1 Formalisms

As discussed in Section 2.2, formalisms are needed in formal methods to define symbolic variables having semantics and mathematical relationships for representing a particular class of target systems. In computer science, FSM-based formalisms are usually represented as Kripke structures [62]:

$$\mathcal{M} = (S, S_0, \rightarrow, L) \tag{3.1}$$

In (3.1), a formal model $\mathcal{M}$ is a four-symbol Kripke structure made up of states ("S") including initial states ("S_0"), transitions to next-states ("$\rightarrow$"), and labels ("L"). The formal model state space is the sum of all states including initial states; transitions are input/output functions mapping a state to a next-state; and labels define the set of valued variables in each state.

Kripke structure representations are defined in a deductive way, starting with high-level states and transitions that are decomposed into lower level symbols having particular semantics [152]. Researchers commonly decompose states and transitions using different symbols to define a formalism that is intended for a particular class of target systems. For example, the hybrid FSM-based formalism in [120] is utilized to define systems having discrete and continuous states. The "S" symbol is partitioned into a finite set of discrete state variables ("Q") and a finite set of continuous state variables ("X").

In human factors, the design process is usually inductive: researchers collect large amount of unstructured data regarding end users and target systems, and the data are utilized to inform the design of prototypes (or computational models) [40, 96]. Thus, in this work formalisms are developed

in an inductive way using the Z specification language (Chapters 4 and 7).

The formal semantics of Z [153] enable FSM-based formalisms to be constructed using hierarchical compositions of schemas, starting with specification of all possible model values and ending with a specification of model outputs. A schema is encoded using basic types, given types, and predicates. Types specify the names and sets of possible values of model variables. Basic types are enclosed in brackets, and their values are not specified within schemas. Given types may be numbers, sets of words, sets of sets, functions, tuples, and types constructed from other types, including schemas.

Given types and predicates of a schema are specified and represented visually within a three-sided box, separated by a line into upper and lower segments. The name of a schema appears within the top-edge of a box. Given types are specified in the upper segment and predicates (if any) are specified in the lower segment. Predicates constrain the possible values of one or more variables using logical rules. For example, in the instantiated schema below named "*example*," the variable named "*number*" has a given type of positive integers (denoted by $\mathbb{N}$). The predicate part (lower half of the box) specifies that its value must be an integer less than 99.

Next-state transitions of an FSM can be represented in Z by encoding a separate schema. Its name should aid in identifying the existing schema whose state variables are transitioning, and its declaration should append a delta (Δ) symbol to the front of that schema's name. Such a declaration introduces all variables declared in the existing schema and a copy of their next-states, which are denoted by adding an apostrophe to the end of their names. For example, the transition schema below specifies transitions to "*example*." Its name is "*next_state_of_example*," and its declaration is $\Delta example$. The predicate specifies that all next-states of the variable named "*number*" (denoted by *number'*) must be greater than 9.

```
next_state_of_example
-------------------------------
Δexample
-------------------------------
number' > 9
-------------------------------
```

Hierarchical schemas can be encoded by instantiating variables having schema types. This technique is demonstrated in Chapter 7.

3.2 Model Checking System

In this work, discrete device, task analytic, and hybrid formalisms are instantiated using the syntax of Symbolic Analysis Laboratory (SAL) [68, 154]. SAL's framework includes an intermediate language for describing transition systems and serves as the target for translators that extract the transition system description. The SAL constructs utilized throughout this document are described in this section.

3.2.1 SAL Contexts

SAL models are defined as named contexts having named types, functions, modules, and theorems. A SAL context is encoded in the form,

```
named_context: CONTEXT =
BEGIN
   named_type: TYPE = ...
   named_function(input_1: named_type,..., input_N: named_type): ...
   named_module: MODULE =
   BEGIN
     LOCAL  variable_1: named_type
     INPUT  variable_2: named_type
     OUTPUT variable_3: named_type
     INITIALIZATION
      variable_1  = ...
     TRANSITION
      variable_1' = ...
     DEFINITION
      variable_3  = ...
   END;
   named_theorem: THEOREM ...|- ...
END
```

Types are defined globally within a context as enumerated lists, numbers, tuples, records, arrays, or types constructed from other types. Enumerated types appear in all subsequent chapters. They are encoded as named sets of comma-separated words or phrases in the form,

```
named_type:  TYPE = {value_0,..., value_N};
```

One type constructed from types are records, which appear in Chapters 6 and 7. They are encoded as named sets of identifier-type pairs in the form,

```
named_type:  TYPE = [#identifier_1:  type,..., identifier_N: type#];
```

Another type constructed from types are arrays, which appear in Chapter 6. They are encoded as named sets of indexed types in the form,

```
named_type:  TYPE = array type of type;
```

Named functions could be encoded using types as inputs and outputs. Outputs could be values having a discrete type, such as the Boolean values *true* or *false*. Outputs could alternatively have a continuous type, such as a decimal-valued number. Like types, functions are declared globally within a context, and they appear in Chapters 8 and 9.

3.2.1.1 The Module Construct

SAL modules describe transition systems in terms of variables and commands, and they appear in all subsequent chapters. Variables can be local, input, and output, and each has a name and type. Local variable values can be accessed and assigned from within a module; output variable values can be accessed by any module and assigned within one; and input variables can be accessed by any module, but values cannot be assigned.

Commands are utilized for accessing and/or assigning variable values. The three kinds of commands in SAL are called initializations, transitions, and definitions, and a set of commands are encoded within modules under a specific heading. Initializations (encoded under the heading `INITIALIZATION`) define one or more variable values in a formal model's initial state; transitions (encoded under the heading `TRANSITION`) define next-state values; and definitions (encoded under the heading `DEFINITION`) define both. Initializations and transitions could be composed of a guard

and a value assignment. The guard is a Boolean expression containing one or more input, output, and local variables as well as SAL's mathematical operators such as AND, OR, and NOT. For transitions, value assignments determine next-states of local and output variables when the guard is satisfied. Next-state values are denoted by an apostrophe character (e.g. value_1'). Guarded transition commands appear in Chapters 4–8, and examples are shown below. Guarded initialization commands assigning initial states instead of next-states appear in Chapter 8.

```
TRANSITION [
 guard_1 -->
  variable' = value_1;
 []guard_2 -->
  variable' = value_2;
   ...
 []guardN -->
  variable' = value_N;
];
```

Value assignments can be encoded for initializations, transitions and definitions in many ways, such as using conditional expressions, equality expressions and selection statements, all of which could be guarded or unguarded. Equality expressions for enumerated type variables (demonstrated in the guarded transition commands above) appear in all subsequent chapters, and they are encoded in the form,

```
variable = value;
```

Equality expressions for record type variables are similar to the encoding for enumerated type variables. Instead of the variable name appearing alone on the left-hand side of an equals sign, the variable and one identifier appears, separated by a period. This kind of equality assignment is shown below, and it is utilized in Chapter 7.

```
variable.identifier_1 = value;
```

For array types, equality assignments can be encoded for individual elements of the array or for all elements. Consider an array named variable_1 that has three Boolean-valued integers as elements. The SAL syntax for assigning individual element values for such an array is shown below, where the variable name and one index within brackets appears on the left-hand side of an equals

sign and a Boolean value appears on the right.

```
variable_1[1] = true;
variable_1[2] = true;
variable_1[3] = true;
```

Instead of utilizing this syntax, an equivalent SAL representation is employed in Chapter 6 for assigning values to all elements of an array. This syntax is shown below for `variable_1`.

```
FORALL(x:  INTEGER): variable[x] = true;
```

Conditional expressions could be encoded for enumerated, record, and individual indexes of array type variables. They work by assigning a particular value if some specified condition holds, where one or more conditions and values could be specified within a single expression. They appear in Chapters 5–7 and are encoded in the form,

```
variable = IF X = 1 THEN value_1 ELSIF ... THEN ... ELSE value_N ENDIF;
```

Selection statements assign a named variable on the left-hand side of `IN` one randomly selected value from a list or function on the right-hand side. Selection statements having lists are utilized in Chapters 4 and 7, and they are encoded in the form,

```
variable IN {value_1, ..., value_N};
```

Selection statements having functions are utilized in Chapter 8, and they are encoded in the form,

```
variable IN function(value_1,..., value_N);
```

Modular model compositions representing interactions among device, user, and environmental elements are encoded within the SAL theorem construct (discussed in Section 3.2.3.1).

3.2.2 Relational Abstraction of Differential Equations in SAL

As discussed in Section 2.4.2.2, continuous device behaviors can be represented by instantiating a hybrid formalism. In this work, the continuous states of hybrid formal models are instantiated using a relational abstraction technique that was originally developed to model differential equations [119]. A differential equation defines continuous changes to an output variable as a function of one or more input variables, and they are often utilized in engineering domains to model continuous system

dynamics. For example, the equation for Newton's law of cooling (3.2) describes an object's change in temperature over time ($\frac{dQ}{dt}$) as a function of the object's current temperature (Q), minus the constant temperature of its surrounding environment (Q_{env}), multiplied by some constant parameter (k) that is always negative.

$$\frac{dQ}{dt} = k(Q - Q_{env}) \tag{3.2}$$

The equation shown in (3.2) cannot be incorporated within model checking analyses as-is, since it expresses a continuous output having infinitely many solutions; however the output could be represented in a discrete way by replacing the continuous derivative $\frac{dQ}{dt}$ with a discrete relational abstraction. Relational abstraction is a technique for representing differential equations in terms of discrete arithmetic expressions [144]. The word *relational* refers to the way output variables are related to input variables that are increasing, decreasing, or constant; and the word *abstraction* refers to how differential equations are represented: rather than encoding the equation in a way that produces mathematically computable solutions (as in many engineering domains), the analyst attempts to represent an approximate set of solutions considered possible and, ideally, as close as possible to the actual solutions [144].

While there are several ways to employ a relational abstraction, the technique employed in this work requires two kinds of SAL models [68]:

1. One representing the differential equations governing continuous system dynamics (called a *plant* model)
2. Another representing constrained sets of desired *plant* outputs (called a *constraints* model)

One way to encode a *plant* model utilizes uninterpreted functions [46]. An uninterpreted function is typically assigned a name as well as one or more input variables on which an output could depend. Name and input variables do not affect an uninterpreted function's output, but they should aid in identifying what the output is and how it is controlled. For example, using SAL syntax, an uninterpreted function representation of (3.2) could be,

```
NewtonCoolingLaw(k: REAL, Q: REAL, Q_env: REAL): [REAL -> BOOLEAN];
```

which reads, "Newton's law of cooling states that an object's change in temperature depends on a constant k, the object's current temperature Q, and the temperature of its surrounding environment Q_{env}, all of which have real (i.e. continuous) number values." The syntax `[REAL -> BOOLEAN]` expresses that the function's output is a real number, but it will be evaluated as a discrete, Boolean (i.e. *true* or *false*) value instead, valued *true* if the output value is in the set of desired values and *false* otherwise. The plant module employing this function could be encoded as shown below (annotated using italic text).

```
plant: MODULE =
BEGIN
  OUTPUT k, Q, Q_env: REAL
  INITIALIZATION
    k = {x: REAL | x < 0}; k is a constant, negative number
  TRANSITION
    Q' IN NewtonCoolingLaw(k, Q, Q_env); The next-state temperature depends on k,
END;                                     the current-temperature, and the temperature
                                         of the environment
```

The *constraints* model defines sets of desired values. In SAL, a *constraints* model could be encoded as a separate module having *plant* model variables as inputs, one or more Boolean variables as outputs and a series of guarded initialization and/or transition commands adding incremental details about one or more uninterpreted function from the *plant* model. Each guarded command sets a Boolean variable to *false* if the uninterpreted function output falls outside a desired range. For example, a SAL module named *constraints* could be encoded for the function `NewtonCoolingLaw` according to the following three conditions:

1. Since the function being constrained is a differential equation concerning changes, an initialization command should assert that all initial states could be considered realistic
2. If the object's environment is warmer than the object, its next-state temperature Q' must be higher than the current temperature Q; i.e. the object must gain heat in the next-state
3. If the object's environment is cooler than the object, its next-state temperature Q' must be lower than the current temperature Q; i.e. the object must lose heat in the next-state

A Boolean variable named *realistic* could be valued *true* if the above conditions hold and *false* otherwise. One initialization command corresponding to the first condition and two guarded transition commands corresponding to the second two conditions could control the value of *realistic* as shown below in a SAL module named *constraints*.

```
constraints:  MODULE =
BEGIN
 INPUT k, Q, Q_env:  REAL
 OUTPUT realistic:  BOOLEAN
INITIALIZATION
   realistic = true;
TRANSITION [
    Q_env > Q AND Q' < Q -->
     realistic' = false;
  []Q_env < Q AND Q' > Q -->
     realistic' = false;
 ];
END;
```

To conduct model checking analyses that only consider conditions considered realistic, *plant* and *constraints* modules could be composed synchronously within a system model using the SAL theorem syntax described in Section 3.2.3.1.

3.2.3 Verification Methodologies

The SAL suite of model checking tools enables analyses suiting formal models and specifications representing different target system elements, such as those incorporating discrete sets of device settings or infinite actuator speeds. Additionally, particular tools are useful if counterexamples could represent long sequences of states versus short ones, and different verification reports provide useful information regarding model diagnostics such as model size and verification time. The tools and techniques employed in this work are discussed in the following sections.

3.2.3.1 The SAL Theorem Construct

Theorems are SAL representations of temporal logic specifications. They are encoded using module compositions and LTL or CTL syntax. Module compositions define how transitions execute within a system model having one or more named modules using composition operators: `[]` denoting asynchronous composition or `||` denoting synchronous composition.

Modules separated by the asynchronous composition operator transition one at a time during

model checking analyses. For example, `module1 [] module2` reads, "module-one transitions first, followed by module-two, or vice-versa." Such a composition is useful if all variables in one module depend on the current-states in another module. In this case, states in one module are updated after all commands in another module have finished executing. This composition is useful for composing models of an end user and device, where user models have outputs representing actions that operate as inputs to the device model [10]. Changes to the device manifest in the next-state, abstracting the device's reaction to a user input that emerges in the immediate future. The example specification encoded below utilizes this composition. It reads, "it is always true that when the end user feeds an input to the device, its output is equal to 'new message' in the next-state."

```
specification:  THEOREM user [] device |-

 G(user_input => X(device_output = new_message))
```

Models separated by the synchronous composition operator transition together during model checking analyses. For example, `module1 || module2` reads, "module-one and module-two transition at the same time." Such a composition is useful if all variables in one module depend on one or more next-states in another module. In this case, module transitions need to be kept in-sync to ensure that commands utilize the correct next- and/or current-states.

A turnstile operator (`|-`) separates a module composition on the left-hand side from a temporal logic specification on the right-hand side. The SAL syntax encoded below reads, "specification is a theorem to be verified in a system model where module-one and module-two transition at the same time, and it asserts that variable_1 always equals value_1."

```
specification:  THEOREM module1 || module2 |- G(variable_1 = value_1);
```

In models that utilize the relational abstraction techniques (Section 3.2.2), plant and constraints models are composed synchronously such that the Boolean variable "`realistic`" and a proposition asserting plant model values must be true at the same time. In the example specification encoded below (based on plant and constraints models of Section 3.2.2), the synchronous composition and implication operator ensure that model checking analyses only consider realistic conditions; i.e., " realistic conditions imply that the object's temperature is always below 80° C."

```
specification:  THEOREM plant || constraints |- G(realistic => Q < 80);
```

3.2.3.2 Symbolic Model Checking

The symbolic model checker utilized in this work (SAL-SMC) [68] is useful for verifying LTL specifications in formal models having finite sets of discrete state variables (CTL specifications are addressed in Section 3.2.3.3 and infinite sets of continuous variables are addressed in Section 3.2.3.5).

Verification is performed automatically by representing a formal model as a binary decision diagram (BDD) [150], a tree-like structure capturing all possible states. Once constructed, the BDD is stored in the computer's random access memory (RAM), and SAL-SMC searches it exhaustively for LTL specification violations using a specialized algorithm (see [155] for details). In addition to returning *proved* or a counterexample, verification reports could include the number of states in the model, time in seconds required for BDD construction, and time in seconds required for verification. BDD construction typically increases with the number of states and transitions in a model, and verification time typically increases with the number of variables and values in the specification. Since the BDD must be stored in RAM, the computer on which the model is analyzed determines whether symbolic model checking is possible. If the computer runs out of RAM before BDD construction or verification completes, model checking fails.

SAL-SMC is particularly useful for verifying models and specifications with counterexample sequences having many steps (explained in Section 3.2.3.4). This capability is leveraged in Chapter 5, where models and specifications could produce lengthy counterexamples. Since SAL-SMC verification reports return the number of states in a model, it is also useful for conducting scalability evaluations for models that are generated automatically using custom translation tools (see for example [156]). Additionally, since SAL-SMC returns the number of steps in lengthy counterexamples, it can be utilized to inform the bounded model checking technique (discussed in Section 3.2.3.5). This capability is leveraged in Chapters 6, 7, and 9.

3.2.3.3 Witness Model Checking

The witness model checker utilized in this work (SAL-WMC) [68] utilizes the same BDD construction mechanism as SAL-SMC. However, verification is conducted in a way that enables the analyst to view *witnesses*: traces through the model leading up to a state in which the specification is proven true. Additionally, SAL-WMC enables verification of CTL specifications. These capabilities are leveraged within model checking analyses in Chapter 4.

3.2.3.4 Bounded Model Checking

Like SAL-SMC, the bounded model checker utilized in this work (SAL-BMC) [68] can be employed to verify LTL specifications in formal models having a finite set of discrete state variables. However, it does not require BDDs; instead, model construction and verification are performed simultaneously using a different kind of algorithm (see [149] for details). Upon invoking SAL-BMC, the formal model's initial state, next ten states, and an LTL specification are converted to a set of conjunctive clauses, e.g. $(a \vee b) \wedge (\neg a \vee b) \wedge \neg p$, where a and b are Boolean variables representing states in the formal model and p is the specification. If every clause is *true*, including the negation of p, the verification report provides a counterexample that has no more than ten steps. Otherwise, it returns "no counterexamples." If desired, the analyst can specify how many steps to consider in a counterexample.

Advantages of bounded model checking over symbolic model checking are the absence of BDD construction times and corresponding RAM requirements, typically enabling faster analyses of larger models. These advantages are leveraged in Chapter 7, but they generally diminish vis-a-vis symbolic model checking of the same model/specification when considering counterexamples with 60 or more steps [149].

3.2.3.5 Infinite-Bounded Model Checking

The infinite bounded model checker utilized in this work (SAL-INF-BMC) [68] utilizes the same verification algorithm as SAL-BMC; however, it also enables model checking analyses of models and LTL specifications incorporating continuous variables and infinite states. This is accomplished using

a decision problem representation from computer science called satisfiability modulo theories (see [157] for more details). Leveraging this advantage, SAL-INF-BMC is invoked in Chapters 8 and 9.

As mentioned, SAL-SMC can be utilized to compute the length of counterexamples in models having discrete state variables, while SAL-BMC and SAL-INF-BMC are, by default, limited to counterexamples having ten steps or fewer. As discussed in Chapter 2, the integrated framework architecture incorporates models having both continuous and discrete state variables, where one or more task models may have counterexamples with more than ten steps. Thus, in Chapter 9, SAL-SMC is utilized to compute the longest possible counterexample with respect to discrete framework sub-models. This result is then leveraged to inform the invocation of SAL-INF-BMC such that an equally lengthy counterexample can be produced.

3.2.4 Verification Apparatus

All model checking analyses in this work are performed on a 3.5 GHz Intel Xeon workstation with 64 GB RAM running the Ubuntu 16.04 LTS desktop.

3.3 Encoding Tools

Modeling techniques for instantiating task analytic formalisms described in Chapters 5–7 do not require the analyst to be familiar with SAL. Instead, they leverage Extensible Markup Language (XML) [71], an international standard for organizing data into a hierarchical-heterarchical node structure.

Custom XML grammars are defined using REgular LAnguage for XML Next Generation (RELAX NG) [158] or XML Schema Definition (XSD) [159], both of which are international standards for describing and constraining the contents of XML documents. An XML document contains a single *root* node and zero or more *child* nodes. Direct children of a node are called *siblings*. Each node may contain text content, valued attributes, and zero or more child nodes. A generic graphical/textual representation of these formal semantics is shown in Fig. 3.1.

One advantage of XML-based grammars is that they support analysts in representing systems formally without the need for model checking syntax. However, XML-based representations are

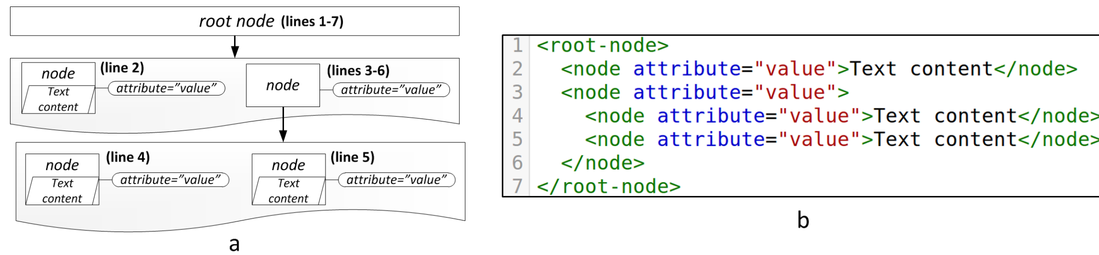


Figure 3.1: A generic XML document with corresponding graphical and textual representations. (a) The graphical representation of XML code utilized throughout this document. Nodes are contained within square-edge rectangles. Attributes are contained within rounded-edge rectangles. Text content is contained within parallelograms. Arrows point from parent to child nodes. Sibling nodes are grouped within shapes. Bold text in parentheses is added to aid in identifying XML line numbers in (b) that correspond to each graphically represented node. (b) XML code represented graphically in (a)

not amenable to model checking analyses. A second advantage of XML addresses this: widely utilized programming languages such as Java and JavaScript have built-in parsing capabilities for reading the contents of an XML document and generating model checking syntax. Leveraging this capability, formal methods researchers have developed a Java-based translation tool (described in [10]) for generating SAL syntax from XML-based representations of human task behavior, which is leveraged in Chapter 5. Inspired by this tool, two new, JavaScript-based translators are developed for different applications in Chapters 6 and 7.

Chapter 4: A Formal Approach to Documentation: Modeling, Specification, and Verification of Navigability [1]

Documentation is part of a safety-critical, human-interactive system interface, and it must be navigable to support time efficiency. Current standards and guidelines for the design of user documentation highlight the need for navigation features that support end users in quickly finding procedures and declarative knowledge required to complete them [29, 19, 161]. ISO/IEC 26514:2008 [29] states that system documentation shall enable users to navigate:

- *back* to return to the section/page visited most recently,
- *next* to the next logical topic/page in the sequence of topics (if any)
- *previous* to the logical topic/page just prior to the one being viewed (if any)
- *up* to the table of contents, top-level menu, or index (if any)

These navigational characteristics are especially important for finding emergency or troubleshooting procedures that must be completed on the order of seconds to minutes. The US Food and Drug Administration (FDA) states in its guidance document for writing medical device patient user manuals, "Format [the troubleshooting] section so the user can locate specific problems quickly" [19]. The US Federal Aviation Administration (FAA) has provided guidance including the suggestion that emergency procedures either be written in a dedicated section of an aircraft flight manual or in a quick reference handbook (QRH) to support quick, easy access [22].

To design documentation that meets these standards, designers generally use navigational tools like labeled page numbers and cross-references. Pages are usually sequentially ordered with numeric labels listed within the top or bottom portion of each page, supporting end users in navigating forward, backward, within, and between sections of the document. Cross-referenced pages are often listed within text, tables of contents, and an index having labeled sections and page numbers (Fig. 4.1

[1] An earlier version of the concepts in this chapter were published in [160]. Figs. 4.1–4.4 and 4.5 and Table 4.1 were published in [160].

and 4.2a). In the case of electronic documentation such as PDF user manuals, hyperlinks can be clicked to navigate between pages within the body of the manual (Fig. 4.2b). For emergency or troubleshooting procedures, designers may include a section in the table of contents for alarm troubleshooting instructions with sub-sections for addressing specific alarms (Fig. 4.1).

5 Alarms 200
Low Battery 202
Power Disconnected 206
Controller Disconnected 214

Figure 4.1: A generic example depicting the structure and function of hyperlinking in the table of contents. Clicking on the section number, 5, navigates the user to page-200. Additionally, clicking on a page number navigates the user to that page

A
accessories 150
Battery Holder 121
Belt Clip 75
Carrying Case 93
Cleaning Cloth 282

a

To temporarily silence the Low Battery alarm, press the *Silence* button ( ) on your controller. **For more information, see Figure 45.**

1. Disconnect the battery cable from your Portable Battery.

2. Remove your Portable Battery from the Battery Holder. For detailed instructions on removing your portable battery from the Battery Holder, see *Battery Holder* on page 121.

b

Figure 4.2: Generic examples depicting the structure and function of hyperlinking in the PDF user manual. (a) Index: Clicking on a page number navigates the user to that page. (b) Main body of the user manual: Clicking "*Fig. 45*" navigates the user to the page containing Fig. 45. Clicking "*Battery Holder* on page 121" navigates the user to page 121

Utilizing labeled page numbers or section titles, end users sequentially turn or rapidly fan through pages in printed documentation to locate necessary content [162]. In electronic documentation, they could scroll through pages using a mouse wheel or utilize "page up" and "page down" keyboard keys to navigate between pages [163]. They may read tables of contents, indexes, and cross-references to learn the location of pertinent content and navigate there by turning multiple pages at once [162]; and in electronic documentation, they could click on hyperlinks to navigate directly [164]. After navigating to a page, end users could remain there for an extended duration while reading, or navigate to a different page if needed, with the intention of returning quickly to the original page.

Currently, analysts have limited tools early in the design cycle for ensuring that navigational tools support these behaviors, and navigable documentation remains elusive in safety-critical systems. For example, in July, 2011 a commercial airline pilot submitted a report to the NASA Aviation Safety Reporting System (ASRS) [161] regarding three potential navigation problems with the QRH (ACN 963587):

1. There was "no way to quickly access checklists" for addressing time-critical emergencies like an engine fire or an auxiliary power unit fire. While there were tab inserts on the QRH, the inserts were not labeled with section numbers or titles.

2. "The index says [the Smoke Removal Checklist] is on page 56 when it is not. 56 has the Smoke, Fumes, Odor checklist, not the Smoke Removal checklist." According to the ASRS report, there is no way to reach the Smoke Removal checklist from its page assignment in the index.

3. "There are certain checklists that inform you to continue to another checklist. It would be helpful if it listed the page number that you have to turn to for the next checklist. With the current system, if one checklist tells you to go to another checklist, the user has to go back to the index to get the page number for the next checklist."

These kinds of problems reflect navigability failures. To inform the design of documentation having time-efficient navigation features, analysts could benefit from a methodology for modeling and verifying navigability. Considering what is needed to support formal modeling and verification, analysts could benefit from an approach providing a formalism, a modeling technique, temporal logic specifications, and a model checking technique. One such approach is provided in this chapter. It includes a formalism for representing documentation and how an end user could navigate through it. A modeling technique is also provided to support the analyst in instantiating the formalism using the syntax of a model checking system. To enable formal verification of documentation navigability, temporal logic specifications and a model checking technique are leveraged from an existing methodology in formal methods [45]. The approach is applied in a case study based on a medical device

PDF user manual, and discussions of case study results, methodological considerations, and future work follow.

4.1 Requirements of a Formalism for Representing Documentation Navigation

The formalisms discussed in Chapter 1 have symbols and semantics for representing end-user tasks, digital control logic, and continuous actuators. To support the analyst in representing a printed or electronic document this work asserts the need to develop a different formalism having symbols and semantics for representing documentation navigation, including:

- Pages of a printed or electronic document
- An end user interacting with the document by viewing content on a page or navigating to a different page

A minimal set of requirements are listed below. The first requirement concerns modeling the document, while the next four concern end user behaviors.

1. A formalism should have semantics for representing sequentially ordered pages

As mentioned, documentation designers often incorporate labeled page numbers that are sequentially ordered to support end users in navigating within and between sections. The formalism should have semantics for representing this common characteristic of documentation.

2. A formalism should have semantics for representing an end user remaining on a page

Pages in documentation could contain instructional procedures, declarative knowledge, and other content that is needed to support safe human-system interaction. End users could stay on a page for as long as needed to review the information. The formalism should therefore have semantics for representing the behavior of remaining on a page.

3. A formalism should have semantics for representing an end user turning to a next-page

If there appears to be unfinished content on a page, such as an incomplete sentence or instructional procedure, end users may remain on the current page or check the next page; thus, the formalism should also have semantics for representing the user turning to a subsequent next-page.

4. A formalism should have semantics for representing an end user navigating to a next-page using cross-references

In printed documentation, cross-references often include page numbers appearing toward the top- or bottom-corners of other pages. End users could attempt to locate these pages quickly by flipping through the document; thus, the formalism should have semantics for representing such a behavior. Electronic documentation has hyperlinking functions introducing cross-references that are not labeled page numbers, such as the *Fig. 45* hyperlink in Fig. 4.2. The formalism should therefore provide a way of representing an end user clicking on these hyperlinks.

5. A formalism should have semantics for representing sets of possible next-pages

Requirements 4.1–4.1 identified the need for semantics that represent an end user remaining on the current page, turning to a next-page, utilizing cross-references, or in the case of electronic documentation, clicking hyperlinks. As shown in Fig. 4.1, there could be multiple cross-references on the same page. There could also be cross-references to content that is useful unfinished content on the same page. The formalism should therefore have semantics for representing sets of next-pages that are possible, one of which can be navigated to via a particular end user behavior.

4.2 Representing Documentation Navigation Formally

To support the analyst in representing documentation navigation formally, the formalism developed in this work aims to satisfy requirements in Section 4.1. In support of Requirement 4.1, its formal semantics capture sequentially order pages in printed or electronic documentation. In support of Requirements 4.1–4.1, next-state transition semantics are provided to represent a set of possible next-pages that can be reached via one of three end user behaviors:

1. Remaining on a page

2. Navigating to a next-page by turning the current one

3. Navigating to a next-page utilizing a cross-reference or hyperlink

To support the analyst in instantiating the formalism, a modeling technique is provided using the model checking syntax of SAL [68]. This modeling methodology is discussed next, including a demonstration using a 4-page user manual.

4.2.1 Formalism

In this work, the formal semantics of a documentation navigation model are represented using two *Z* [153] schemas (see Section 3.1 for details about *Z* notation):

1. *documentation*: specifies a set of integers representing numbered pages in a printed or electronic document

2. *navigation*: specifies transitions and a set of next-pages, where one next-page is possible via the end user remaining on a current page, turning the current page one forward, or navigating to a cross-referenced page

These schemas specify a document in terms of three page types:

1. Pages having content

2. Pages having navigational tools

3. Pages having both

4.2.1.1 Documentation Schema

In this work, the documentation schema specifies pages in a printed or electronic document. It includes a basic type *[max]*, which is a positive integer representing the last page in the document. The given type *page* represents the current page, which is an integer (denoted by the symbol $\mathbb{Z}$). If pages are numbered using labels that are not integers, such as the decimal representations utilized in the aircraft QRH discussed earlier (e.g. "*5.6*"), these numbers should be abstracted as integers.

The first predicate specifies that *page* must be less than or equal to max (i.e. the user cannot navigate beyond the last page), while the second predicate specifies that *page* must be greater than or equal to 0 (i.e. page numbers cannot be abstracted as negative integers).

documentation

$page : \mathbb{Z}$

$page \leq max$

$page \geq 0$

4.2.1.2 Navigation Schema

In this work, the *navigation* schema specifies three kinds of next-page transitions representing the three end user behaviors considered in Requirements 4.1–4.1. The declaration $\Delta documentation$ specifies that this schema represents changes to the *documentation* schema; i.e., next-state transitions of the variable *page*. The functions *keepPage*, *turnPage*, and *crossRef* represent the three respective end user behaviors of remaining on the current page:

1. *keepPage*: In printed or electronic documentation, this behavior manifests as reviewing a page having tables, diagrams, or multiple lines of text

2. *turnPage*: In printed documentation, this behavior could manifest as turning one page. In electronic documentation, this behavior could manifest as scrolling to the next page, pushing a “page down” keyboard key (if any), or clicking an icon in the electronic reader application having a similar functionality (if any).

3. *crossRef*: In printed documentation, cross-references are page numbers or section titles that are printed toward the top or bottom corners of pages. This behavior could manifest as the end user fanning through the document to locate a page quickly. In electronic documentation, cross-references can be hyperlinks that the end user can click to navigate to a page directly. Cross-references that are not working hyperlinks are not represented, as navigating to them via scrolling or repeated keystrokes is considered a set of multiple actions.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{navigation}} \\
\Delta \textit{documentation} \\
\textit{keepPage} : \mathbb{Z} \mapsto \mathbb{Z} \\
\textit{turnPage} : \mathbb{Z} \mapsto \mathbb{Z} \\
\textit{crossRef} : \mathbb{Z} \mapsto \mathbb{Z} \\
\hline
\textit{keepPage}(\textit{page}) = \{p : \textit{page}' \mid p = \textit{page}\} \\
\textit{turnPage}(\textit{page}) = \{p : \textit{page}' \mid p = \textit{page} + 1 \\
\forall\, r : \textit{page} \bullet \textit{crossRef}(r) = \{p : \textit{page}' \mid p = r\} \\
\forall\, p, r : \textit{page} \bullet \\
\quad \textit{page}' \in \textit{keepPage}(p) \cup \textit{turnPage}(p) \cup \bigcup_{i=1}^{n} \textit{crossRef}_i(r)
\end{array}
$$

All three functions are a one-to-one mapping between two integers ($\mathbb{Z} \mapsto \mathbb{Z}$), meaning they take one page number integer as an input and provide one page number integer as an output; however, their semantics differ. The first predicate specifies the semantics of *keepPage*. It states that if the user wishes to stay on the current page ($\textit{keepPage}(\textit{page})$), then the next-page does not change ($p : \textit{page}' \mid p = \textit{page}$, where $'$ denotes "next"). The second predicate specifies the semantics of *turnPage*, which states that the next-page is one-integer greater than the current-page ($p : \textit{page}' \mid p = \textit{page} + 1$). The third predicate specifies the semantics of *crossRef*. It states that for all cross-referenced pages on the current page ($\forall\ r : \textit{page}$), one page can be navigated to ($p : \textit{page}' \mid p = r$. The fourth predicate specifies that for all pages and cross-referenced ones ($\forall\, r : \textit{page}$), the next-page ($\textit{page}'$) comes from a set of possible next-pages than could be navigated to via one of three behaviors ($\in \textit{keepPage}(p) \cup \textit{turnPage}(p) \cup \bigcup_{i=1}^{n} \textit{crossRef}_i(r)$, where n is the number of cross-referenced pages).

4.2.2 Modeling Technique

To instantiate the formalism, the analyst should first calculate the number of pages in the document, starting from 0. For example, if a document has ten pages of front matter labeled i–x in Roman numerals, 20 pages of body material labeled 1–20, and five pages of back matter without labeled page numbers, there are 35 pages in the model numbered 0–34. Using the semantics defined in Section 4.2.1, this corresponds to assigning the basic type *[max]* a value of 34. Next, the analyst should go through each page of the document and determine what end user behaviors are plausible with respect to the functions defined in Section 4.2.1. This technique can be applied using the model checking syntax of SAL [68] (see Chapter 3, Section 3.2 for explanation of SAL syntax). It is

demonstrated below for a 4-page example document having:

- A table of contents on page-0 with cross-references to pages-2 and 3
- A procedure that begins on page-2 and continues on page-3
- A cross-reference on page-3 back to the table of contents

Italic text is added to aid in identifying what SAL syntax corresponds to schema elements of Section 4.2.1. Boldface text is added to explain the semantics of guarded transitions and selection statements.

```
documentation:  CONTEXT =
BEGIN
 keepPage(page:  INTEGER): INTEGER = page;      "keepPage" of navigation schema
 turnPage(page:  INTEGER): INTEGER = page + 1;  "turnPage" of navigation schema
 crossRef(ref :  INTEGER): INTEGER = ref;       "crossRef" of navigation schema

  navigation:  MODULE =
  BEGIN
   OUTPUT page:  {x:  INTEGER | x >= 0 AND x <= 3} "page" of documentation schema,
                                                  where the basic type "[max]" is 3

   TRANSITION [ fourth predicate of navigation schema, beginning "∀ p : page; r : page •"

   page = 0 -->                                From the table of contents (page-0),
    page' IN {crossRef(2), crossRef(3)}; the user can navigate to page-2 or 3.

   []page = 2 -->                                     From page-2, the user can stay
    page' IN {keepPage(page), turnPage(page)}; or turn to the next page

   []page = 3 -->                                  From page-3, the user can stay
    page' IN {keepPage(page), crossRef(0)}; or navigate to page-0
  END;
END
```

4.2.3 Specifications

The ASRS report discussed earlier highlights potential page reachability failures in the aircraft QRH. In support of interface completeness, reachability specifications have been developed in formal methods for identifying navigability-related problems that could emerge for end users of displays and controls:

1. Weak task connectedness [45]: Starting from any state, there is at least one way for the user to complete a procedure

2. Weak task completeness [45]: Starting from an initial state, the user can eventually complete a procedure
3. Reversibility [45, 51]: The effects of an action can be undone in one action

In the context of documentation navigation, this work leverages them to provide a set of page reachability specifications that assert time-efficiency of navigational tools:

1. Page connectedness: From any page, the user can eventually reach a goal page.
2. Navigation completeness: From any page, the user can always reach a goal page.
3. Cross-reference reversibility: After navigating away from an initial page, there is at least one way for the user to return back to that page in one step.

To support model checking analyses, this work encodes these specifications using computational tree logic (CTL) as shown in Table 4.1.

Table 4.1: Computational tree logic (CTL) representation of page reachability specifications

Specification	CTL	Interpretation
Weak page connectedness	$\mathsf{EF}(page = goal)$	From any page, the user can eventually reach a goal page.
Weak navigation completeness	$\mathsf{EF}(page = initial \wedge \mathsf{EF} page = goal)$	From an initial page, the user can eventually reach a goal page.
Cross-reference reversibility	$\mathsf{EF}(page = initial \wedge \mathsf{EX}(\neg(page = initial)) \wedge \mathsf{EX}(\mathsf{EX}(initial)))$	After navigating away from an initial page, there is at least one way for the user to return back to that page in one step.

Utilizing these specifications, Fig. 4.3 shows examples of navigability successes and failures. The bottom row shows a hypothetical version of an aircraft QRH that has navigability problems with respect to time-efficiency. The upper row demonstrates a potential improved design that supports navigability.

4.2.4 Modeling Checking Technique

Specifications are verified utilizing the SAL witness model checker (SAL-WMC) [68], which enables verification of CTL specifications. If violations are detected, a SAL-WMC generates a counterexam-

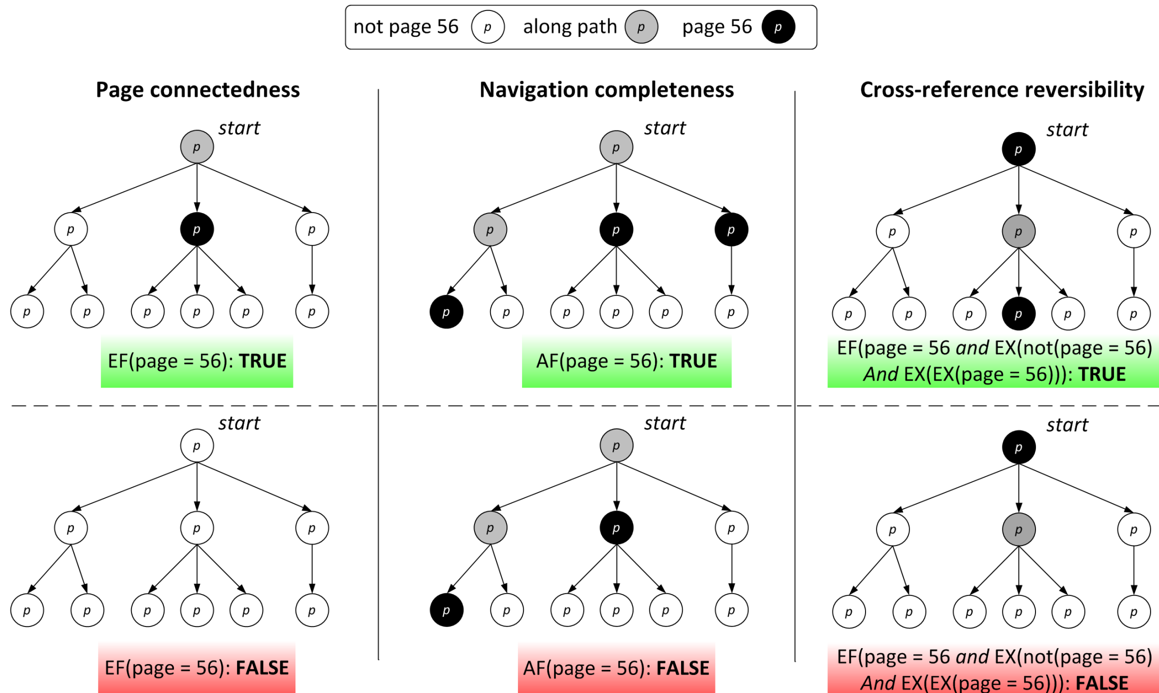


Figure 4.3: CTL specifications and path trees representing paths through different possible designs of an aircraft QRH

ple showing a trace through the model leading up to the violation. Otherwise, SAL-WMC returns a witness as a trace through the model leading up to a state satisfying the specification.

4.3 Case Study: A Medical Device PDF User Manual

A case study inspired by a medical device PDF user manual was developed to demonstrate the approach. The case study manual has 305 pages, 263 of which contain material that is relevant to setup, operation, maintenance, or troubleshooting of the device. Navigational tools include a main table of contents, several sub-tables of contents on the first page of every section, an index, and hyperlinking functions.

The case study device has a controller that can be connected to either an AC power supply (i.e. a wall outlet) or a portable battery. The portable battery can be secured in a holster connected to a canvas belt that is worn around the end user's waist. If the controller is connected to a portable battery, one of two alarms can sound: one indicating that the battery is completely discharged or one indicating that the battery's charge level is low. If the discharged battery alarm sounds, the end

user must navigate to page-66, which contains an instructional procedure for switching the power supply from a portable battery to AC power. If the low battery alarm sounds, the end user must navigate to page-202 and follow the instructional procedure for replacing it. In such a situation the end user needs to know how to remove a portable battery from its wearable holster. This requires him to navigate from page-202 to page-121, which contains a different procedure for removing the battery from the holster, and then back to page-202 to complete the battery replacement procedure.

When the controller is connected to either AC power or a portable battery, an alarm sounds to alert the user that a power supply is disconnected. In such a situation, the end user must navigate to page-210, which contains an instructional procedure for reconnecting a disconnected power supply.

For the purpose of analyzing navigability, cross-referenced pages that can be reached in one step by clicking hyperlinks are considered in the model. Pages containing procedures enable the end user to remain on the current-page. Pages containing unfinished sentences or additional procedural steps that continue onto the next-page enable the end user to navigate one-page forward.

4.3.1 The Formal Model

To represent navigation through the case study manual, the documentation navigation formalism was instantiated in SAL. To discover hyperlinked pages that should be included in the model, a search for hyperlinked pages, tables and figures was conducted by searching the PDF for words "page," "table" and "figure." Hyperlink locations were verified using the *AutoBookmark* plugin tool from EverMap LLC, which generated a report of all navigational links within the PDF document (Fig. 4.4).

This yielded 152 lists of reachable pages, 30 of which are relevant to the case study. Page-0 was modeled as the table of contents. All 152 pages with one or more reachable-next-pages (13 in total) were each assigned a respective transition statement in SAL:

1. Page-0 (the main table of contents) containing hyperlinks to pages 11, 66, 101, 178, 200, 202, 206, 214, and 225

2. Page-11 containing a sub-table of hyperlinks to pages 22 and 140

```
AutoBookmark Plug-in Link Report

Report date: Tuesday, February 24, 2015, 14:45:49
Document name: C:~/../Patient_Handbook.pdf
Total number of pages checked: 305
Total number of links found: 659
  ...
```

```
Links Statistics:

Total number of absolute path links: 0
Total number of links pointing to a page in the
same document: 805
  ...
```

```
Errors Found:
  ...
Total number of links without actions: 90
 [1] Page 25, Error: Link annotation does not
have an action.
  ...
 [90] Page 238, Error: Link annotation does not
have an action.

END OF REPORT
```

Figure 4.4: A fragment of the link report generated by the EverMap LLC *AutoBookmark* plugin tool [11]

3. Page-66 containing a procedure and hyperlinks to pages 93 and 99
4. Page-70 containing a sub-table of contents having hyperlinks to pages 22, 88, and 263
5. Page-75 containing a procedure that continues onto page-76
6. Page-101 containing a sub-table of contents having hyperlinks to pages 93 and 99
7. Page-200 containing a diagram and a hyperlink to page-210
8. Page-202 containing a procedure that continues onto page 203 and a hyperlink to page-121
9. Page-203 containing a labeled diagram and hyperlink to page-121
10. Page-206 containing a procedure that continues onto page-207
11. Page-207 containing remaining procedural steps continued from page-206, an unfinished sentence that continues onto page-208, and a hyperlink to page-210
12. Page-210 containing a procedure and hyperlinks to pages 121 and 214

13. Page-225 containing a sub-table of contents having hyperlinks to pages 75 and 140

After all transition statements were encoded in SAL, the model was assigned an initial state of *page = 0*. The final model is 43 lines of SAL code (Appendix D.1).

4.3.2 Specifications

User manual navigation scenarios addressed the CTL reachability specifications as described in Table 4.1. They are encoded to assert navigability in three scenarios:

1. The discharged battery alarm sounds, requiring the end user to navigate to page-66 and follow the instructions for switching the power supply from a discharged portable battery to AC power
2. An alarm indicating a disconnected power supply sounds, requiring the end user to navigate to page-210 and follow instructions for reconnecting a disconnected power supply cable
3. The low battery alarm sounds, requiring the end user to navigate to page-202. From page-202, the end user must navigate to page-121 for further instructions on removing the portable battery from its wearable holster; then, the user must navigate back to page-202 from page-121 to complete the battery replacement procedure

4.3.2.1 Weak Page Connectedness

Consider the first scenario identified in Section 4.3.2: the discharged battery alarm engages and the user needs to follow the procedure for switching from a discharged portable battery to AC power. In such a situation, page-66 containing the necessary instructions should be navigated to quickly. To assert that the end user can locate this page, the specification for *weak page connectedness* in (4.1) reads, “the end user can eventually reach page-66.”

$$\mathsf{EF}(page = 66)) \tag{4.1}$$

4.3.2.2 Weak Navigation Completeness

Consider the second scenario identified in Section 4.3.2: the cable connecting a power supply to the controller becomes disconnected. In such a situation, page-210 containing instructions for reconnecting the disconnected cable should be navigated to quickly. For the purpose of demonstrating a case of *weak navigation completeness*, these instructions should be accessible from the table of contents. Thus, the specification for *weak navigation completeness* in (4.2) reads, "starting from the table of contents, the end user can eventually reach page-210."

$$\mathsf{EF}(page = 0 \wedge \mathsf{EF}(page = 210)) \tag{4.2}$$

4.3.2.3 Cross-Reference Reversibility

Consider the third scenario identified in Section 4.3.2: the low battery alarm engages. In such a situation, the end user must navigate to page-202 containing instructions for addressing the low battery alarm, and further instructions are needed for the end user to remove the portable battery from its wearable holster (listed on page-121). Using the hyperlink in the manual, she can navigate away from page-202 to page-121 with the intention of returning to page-202 once the supplemental procedure on page-121 is complete. This scenario can only lead to a successful outcome if page-121 contains a navigational access point back to page-202. To assert that this should be possible, the specification for *cross-reference reversibility* was encoded as shown in (4.3). It reads, "there exists a path starting from page-202 where if the user navigates away from page-202, then she can find her way back to page-202 in one step."

$$\mathsf{EF}(page = 202 \wedge \mathsf{EX}\neg(page = 202)) \wedge \mathsf{EX}(\mathsf{EX}(page = 202)) \tag{4.3}$$

4.4 Verification

Reachability specifications were verified using the SAL witness model checker (SAL-WMC) [165]. Results are reported in Table 4.2. As mentioned, when the specification is proven valid, the model checker provides a witness as a path through the user manual that satisfies the specification. Reported witnesses were validated by physically enacting them using the original PDF user manual. For violated specifications, SAL-WMC generated a list of reachable-next-pages as starting points for counterexample paths. Any counterexamples generated for invalid specifications were validated by inspecting the original PDF user manual.

Table 4.2: Case study model checking results

Specification name	Result	Execution time (s)
Weak page connectedness	*invalid*	0.06
Weak navigation completeness	*valid*	0.02
Cross-reference reversibility	*invalid*	1.20

Verification of *weak page connectedness* addressed getting to a goal page from any initial page. For verifying the specification, the initial state *page = 0* was removed so the model could assume any of the 263 pages. This was necessary to ensure that the model did not begin its search on a specific page. Passing the model and specification (4.1) to SAL-WMC returned *invalid*. This analysis showed that there is no way of getting to page-66 using hyperlinks (represented graphically in Fig. 4.5a).

Verification of *weak navigation completeness* addressed getting to a goal page from the table of contents. For this specification, the initial state of *page = 0* was reinstated to ensure that the end user starts from the main table of contents. Passing the model and specification (4.2) to SAL-WMC returned *valid*. This result "proves" that starting from the table of contents, there is a way for the user to get to page-210 that lists troubleshooting instructions for reconnecting a disconnected power supply cable. Fig. 4.5b shows one successful path to page-210 from the table of contents.

Verification of *cross-reference reversibility* addressed navigating away from an initial page, and

then returning back to it in one step. Passing the model and specification (4.3) to SAL-WMC returned *invalid*. This result "proves" that once a user navigates away from page-202 there is no way to return to page-202 in one step. Fig. 4.5c shows that there is no path back from page-121. This result represents a reachability failure identified in the model.

4.5 Discussion

This chapter presented a novel application of formal methods to uncover potential time-efficiency problems with respect to documentation navigability. The work presents minimal requirements for a documentation navigation formalism that captures how people have been observed to interact with a printed or electronic document. The formalism developed to meet these requirements provides a generalizable way of specifying sets of next-pages reachable from a current page. The work described a modeling technique for instantiating the formalism in the model checking syntax of SAL, and CTL

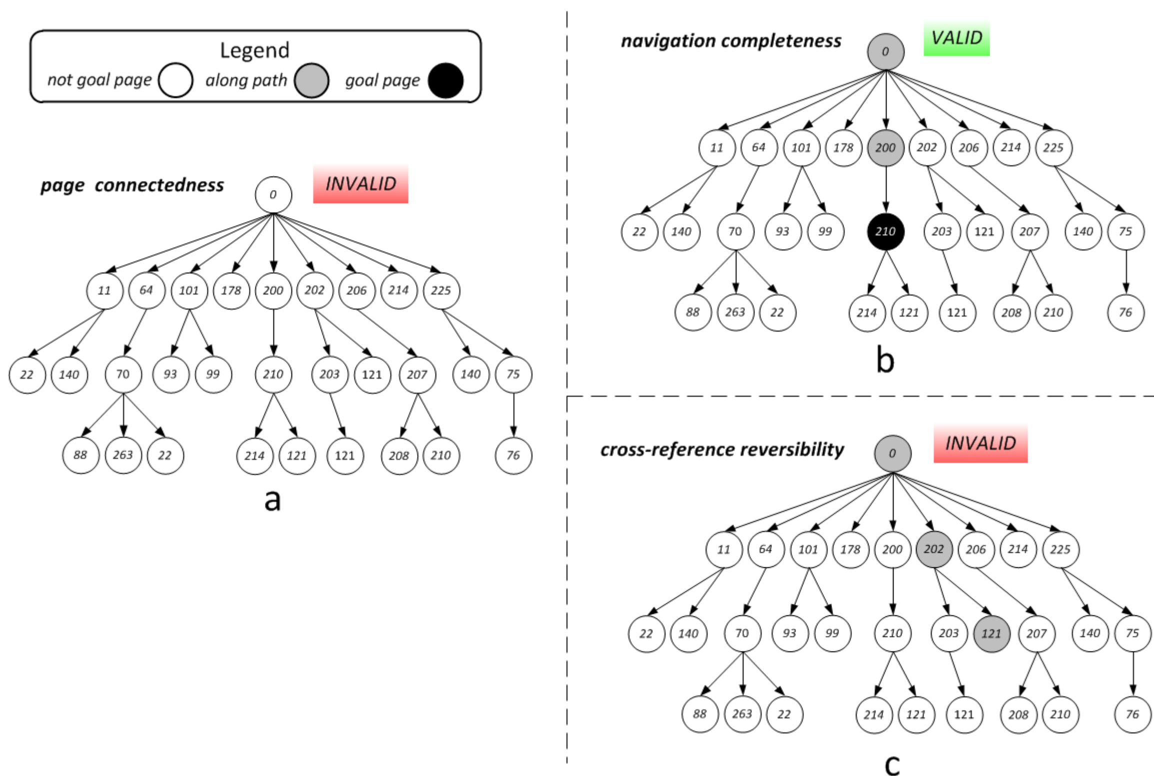


Figure 4.5: Tree representations of the witnesses provided in verification reports generated by SAL-WMC: (a) invalid result for *weak page connectedness*. (b) Valid result for *weak navigation completeness*. (c) Invalid result for *cross-reference reversibility*

page reachability specifications were developed to enable formal verification of documentation navigability. Using witness model checking, the verification technique provides a way of identifying potential time-efficiency problems via counterexample, while witnesses could provide evidence that navigational tools enable time-efficient page reachability. The case study results indicate that navigation problems involving hyperlinks in electronic documentation can can be uncovered using CTL. By supporting formal methods-based analyses of documentation navigation, these contributions extend the capabilities of extant tools and techniques.

4.5.1 Methodological Considerations

The documentation navigation formalism abstracts the pages of a printed or electronic document and document navigability as three functions representing a subset of possible end-user behaviors:

1. Staying on a current-page when there is useful content therein
2. Turning one-page forward when there is unfinished content on the current-page, such as an incomplete sentence or procedure
3. Navigating directly to a page that is cross-referenced on the current-page

Case study results indicate that modeling these three behaviors could be sufficient for uncovering potential time-efficiency problems with respect to a document's navigational tools. However, users may always look at the next-page or previous page, regardless of what is on the current page. While one could add formalism infrastructure representing these behaviors, such a model would not necessarily provide insight to the analyst with regard to time-efficiency concerns: the modeled user could navigate page-by-page through the entire document, and specifications of *weak page connectedness* and *weak navigation completeness* would never be violated. The behaviors modeled in this work therefore appear sufficient with respect to the current set of page reachability specifications; however, modeling a broader set of end-user behaviors could necessitate additional specifications.

4.5.2 Future Work

Areas of future work include improvements to the modeling and verification methodologies. In regard to the modeling methodology, the formalism currently needs to be instantiated manually. In other

human-interactive system applications, researchers have developed encoding tools that automate the model development process, such as the custom grammar and translator in [10]. A similar tool could be developed to automate the process of instantiating the documentation navigation formalism; and for electronic documents, it could be useful for the tool to exclude broken hyperlinks from translated models using an automated plugin such as *AutoBookmark* [11].

In regard to the formalism, a broader range of end-user behaviors should be explored, such as scrolling through multiple pages of an electronic document to reach content that is located several pages from the current-page. It could also be useful to add infrastructure that accounts for the number of end-user actions that are needed to reach a goal page. Such an improvement could enable the analyst to encode specifications asserting that a goal page is reachable within a desired number of steps (e.g., $\mathsf{EF}(page = goal \wedge actions \leq 5)$, meaning, "the end user can eventually reach a goal page in five actions or fewer"). Infrastructure for specifying semantic content of each page could also be useful. Such an improvement would enable the analyst to assert content reachability within navigability specifications, e.g., $\mathsf{EF}(page = troubleshooting)$, meaning, "the end user can eventually reach the page containing troubleshooting instructions." One way of incorporating semantics in this way is explored in Chapter 9.

In regard to the verification methodology, encoding CTL specifications is challenging. A technique for generating them automatically has been developed for use in human-automation interaction [57]. In future extensions to this work, reachability specifications could be automatically generated with respect to users always being able to find the instructions, always being able to go back to where one was, and being able to get to information in a specified number of steps (or page transitions). SAL-WMC verification reports are generated in text format, and interpreting them can be challenging. Visual representations like the path trees illustrated in 4.5 must be generated manually, and future work should explore an automatic verification report visualization tool, similar to the one described in [73].

Chapter 5: A Formal Approach to Documentation: Modeling, Specification, and Verification of Procedures[1]

Chapter 4 highlighted the importance of navigational tools within user documentation that support end users in quickly locating pages. Navigable documentation is critical to time efficiency of the interface; however, it is also critical that procedures are usable. ISO/IEC 26514:2008 [29] states that documentation procedures should be easy for inexperienced end users to understand. In support of accuracy, text should sufficiently describe what components are involved in procedural steps. In support of completeness, it states that instructions should be applicable to all system configurations that are relevant and possible. The U.S. FDA guidance for medical device documentation reiterates these recommendations; additionally, it states that time-critical procedures such as troubleshooting should have logically ordered steps, and that the content should explain how to complete steps with a reduced need for technical knowledge [19].

Currently, analysts have limited tools for ensuring that procedures in documentation are written in an accurate, complete, and time-efficient way, and problems often emerge for end users. For example, during the first six months of 2016 there were 635 medical device adverse event reports in a U.S. national databased all describing the same use-related problem for a home-use dialysis machine:

- During setup, patients were routinely connecting themselves to the device and initiating therapy without properly preparing (or “priming”) a fluid-filled connection tube, causing an alarm to engage
- When attempting to troubleshooting the alarm, users were routinely disconnecting and reconnecting themselves from the device without properly sterilizing the outputs ends of connection tubes, sometimes causing injury [83]

To address these kinds of problems, a new approach is needed early in the design cycle to

[1] An earlier version of the concepts in this chapter were published in [143]

support analysts in ensuring procedures are usable. A modeling methodology should enable analysts to represent what actions an end user could execute based on the content in documentation. A verification methodology should enable analysts to identify potential problems and improvements. One such approach is developed in this chapter.

Leveraging an extant formal task modeling framework [10], which provides a formalism and an encoding tool, a new task modeling technique enables the analyst to represent one or more end-user behaviors that are possible based on what components and parts are identified. This technique intends to support the analyst in identifying accuracy-related usability problems while attempting to represent the procedure formally (discussed further in Section 5.1). The result could be leveraged to inform improvements to the procedure with respect to accuracy.

A device modeling technique provides a way of representing and identifying:

- Initial component configurations that are possible when a procedure begins executing
- Initial component configurations that are addressed by actions prescribed in the procedure
- A potential completeness-related usability problem that could arise if a subset of initial configurations are addressed

The result could be leveraged to inform improvements to the procedure with respect to completeness.

A verification methodology includes LTL specifications and a model checking technique. Specifications assert undesired temporal orderings of procedural steps, and the model checking technique involves verifying specifications with respect to two models of the procedure:

1. One representing the end user executing all procedural steps in order
2. One representing the end user executing one or more procedural steps in any order

If the model checker returned *proved* in the first model, there could be a time-efficiency problem, and the specification can be checked again in the second model. A counterexample returned in the second model could reflect an improved temporal ordering of procedural steps with respect to the specification, and this result could be leveraged to improve the procedure with respect to time efficiency.

A case study based on a medical device troubleshooting procedure is used to demonstrate the approach, and two versions of the procedure are compared using model checking. The first version considers a model representing the procedure exactly as it is written, including the prescribed ordering of procedural steps. The second version considers an alternative task ordering that enables steps to be executed in any order. Discussions of case study results, methodological considerations, and directions of future work follow.

5.1 Modeling Methodology: Representing Procedures in Documentation Formally

The approach for modeling and verifying documentation procedures has two steps:

1. An augmented formal task modeling technique for representing procedures in documentation
2. An augmented discrete device modeling technique for representing initial and functional configurations of configurable hardware

The Enhanced Operator Function Model (EOFM) framework [10], which provides a formal description language, modeling technique, and tool facilitating the development of human-system interaction models, facilitates the first step. The EOFM language's XML syntax has been defined using the RELAX NG [158] standard, which specifies the keywords and structure utilized to instantiate a formal task model as an input/output system [10]. Inputs may come from the device, the task environment, as well as the end user's goals. For example, in a medical context, the goal could include the specific prescription [166] or the preferred operating range of a medical device [56]. Output variables are human actions. Using a hierarchical and heterarchical structure, activity descriptions specify how human actions may be generated based on input and local variables (representing perceptual or cognitive processing). All variables are defined in terms of constants, user defined types, and basic types.

Multiple activities can be defined in the same instantiated EOFM-XML model. Activities have associated strategic knowledge defined by preconditions, repeat conditions, and completion conditions (Boolean expressions written in terms of input, output, and local variables as well as constants)

that specify what must be true before an activity can execute, when it can execute again, and what is true when it has completed execution respectively. Activities are decomposed into lower-level sub-activities and, finally, actions. Decomposition operators specify how many sub-activities or actions can execute and what is the temporal relationship among them (Table 5.1). Actions are either an assignment to an output variable (indicating an action has been performed) or a local variable (representing a perceptual or cognitive action).

Table 5.1: Decomposition Operators [10]

Operator name	Semantics	Modality	
		Sequential	Parallel
and	All sub-activities or actions must execute	*and_seq*	*and_par*
or	One or more sub-activities or actions must execute	*or_seq*	*or_par*
optor	Zero or more sub-activities or actions must execute	*optor_seq*	*optor_par*
xor	Exactly one sub-activity or action must execute	*xor*	—
ord	All sub-activities or actions must execute in order of appearance	*ord*	—
sync	All sub-activities or actions must execute at the same time	—	*sync*

The structure of an instantiated EOFM-XML model can be visually represented as a treelike graph structure using the graphical notation or custom visualizer described in [73] (generic example depicted in Fig. 5.1). Activities not at the lowest level begin with *a* for activity and are surrounded by rectangles with rounded edges. The lowest level activities in the tree begin with *h* for human action and are represented inside rectangles with right-angle edges. Preconditions (conditions for when an activity can initiate) are denoted by yellow triangles pointing downward. Completion conditions (conditions for when an activity has been completed) are denoted by pink triangles pointing upward. Input variables, which begin with a lowercase *i*, can be used in preconditions and completion conditions to govern activity and action execution.

The custom translator generates a task model encoded using the EOFM-XML language directly into the model checking syntax of SAL based on the task analytic formalism described in [10]. This translation process introduces additional model infrastructure [167] required to ensure that human

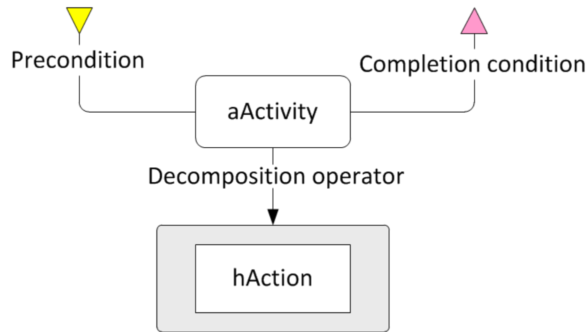


Figure 5.1: Graphical depiction of a generic EOFM-XML activity. "Decomposition operator" can be replaced with any of the keywords shown in Table 5.1. "Precondition" and "Completion condition" can be replaced with valued input variables representing states of the device or operational environment

operator actions can be properly recognized by a device model. The analyst can manually encode device model infrastructure in a way that abstracts human-device interaction via the exchange of input/output variables (discussed in Section 5.1.2).

5.1.1 Task Modeling Technique

Formal task modeling techniques usually produce a normative representation of human behavior reflecting correct performance of goal-driven tasks [96]. To support the analyst in identifying potential accuracy problems, the technique developed in this work centers on end user behaviors that are possible based on how components involved in tasks are described. Specifically, the technique provides a way of encoding procedural steps in which content can be interpreted by the end user in a way that informs different combinations of possible behaviors. For example, consider a hypothetical system having a controller, portable batteries, a battery cable having two connectable/disconnectable output ends, and the following user manual procedure for replacing a low battery:

If the audible low battery alarm engages:

1. Press any button to silence the alarm
2. Disconnect the battery
3. Retrieve a fully charged replacement battery
4. Connect it to the battery cable

For such a system and procedure, the end user could interpret these task descriptions in many ways. If there are many buttons on the controller, any one of them could be pushed; and because the battery cable has two outputs ends, either one (or both) could be disconnected in any order. Leveraging the decomposition operators of EOFM-XML (Table 5.1), the analyst could encode and model many possible behaviors having different effects if a specific component or part is not identified in a task description. In light of the need for safety-critical system procedures to have accurate task descriptions, the purpose of this technique is to aid the analyst in identifying potential accuracy problems while attempting to represent them formally. It proceeds as described in outline form below. Fig. 5.2 serves as a visual aid:

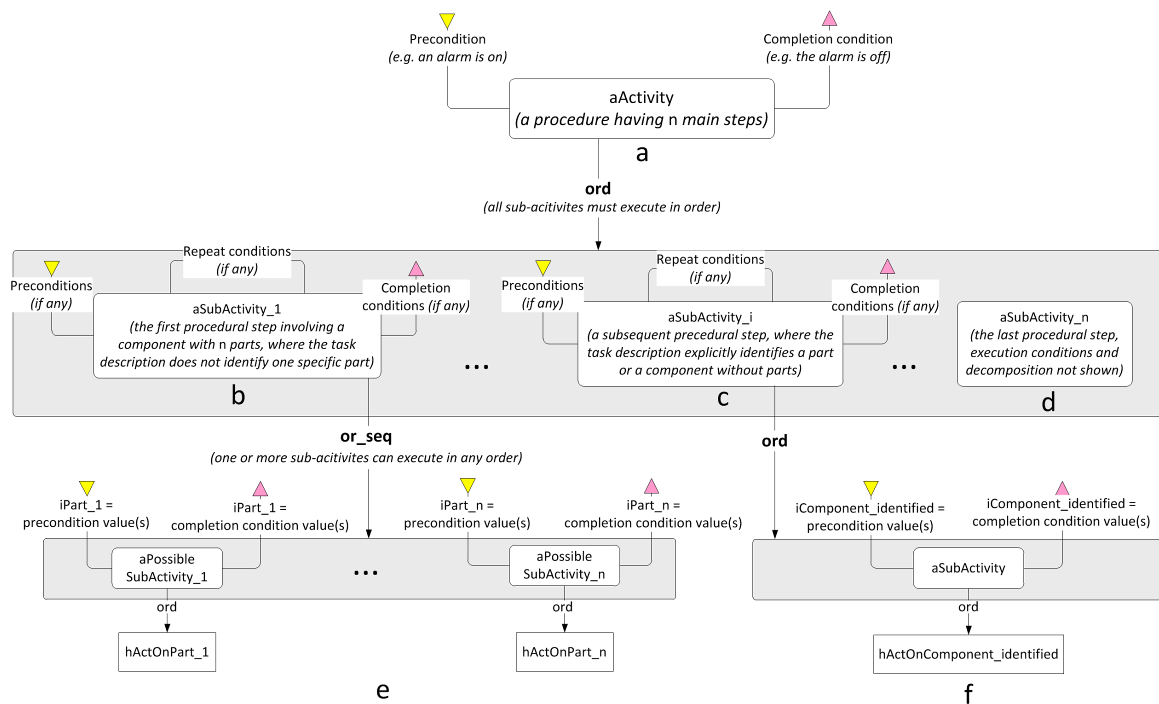


Figure 5.2: Generalizable, graphical representation of the task modeling technique for representing a procedure in accompanying documentation. Annotations in italic text are not part of EOFM-XML's formal semantics. Letters a–f are added for reference in the outlined description of this technique

1. Encode a top-level EOFM-XML activity with precondition(s) defining when the procedure should begin executing; for example, if text on the page indicates that the procedure is for troubleshooting a particular alarm engaged on the device, the top-level precondition should specify what alarm is engaged (Fig. 5.2a)

2. Decompose the top-level activity using the *ord* decomposition operator, specifying that procedural steps must be performed in order (Fig. 5.2a), and encode all heterarchical steps as sub-activities (up to step-n, corresponding to *aSubactivity_n* in Fig. 5.2d)

3. Starting with the first procedural step (and repeating until all steps have been encoded), determine if it involves a component having multiple parts. If so:

   (a) Determine whether the part(s) on which the user should act as well as the temporal ordering of actions are identified explicitly in the content describing it. For example, the text "disconnect the battery cable" could refer to one or both parts being acted upon (i.e., disconnected) in any order; the text "disconnect both ends of the battery cable" refers to both parts being acted upon in any order; and the text "disconnect the output end connected to the battery first, followed by the other end" refers to both parts being acted upon in a specific order. In the first case (neither specific parts nor a temporal ordering of actions are identified):

      i. Decompose the sub-activity representing this step into one or more sub-activities involving each individual part that could be acted upon utilizing the *or_seq* decomposition operator (as in Fig. 5.2b). Such a decomposition represents the end user acting on one or more parts in any order. The analyst should also include task execution conditions to identify when the activity begins executing, repeats executing, and is complete (encoded generally as "Preconditions *(if any)*," "Repeat conditions *(if any)*," and "Completion conditions *(if any)*" in Fig. 5.2)

      ii. Encode sub-activities representing the end user acting on parts 1… n, including preconditions and completion conditions that are necessary (Fig. 5.2c). If there are sub-steps (or sub-sub-steps), each sub-activity should be decomposed accordingly into sub-activities using the same technique. There is no limit to the number of times a sub-activity can be decomposed. Otherwise, each sub-activity should be decomposed into a corresponding human action (encoded generally as *hActOnPart_1* … *hActOnPart_n* in Fig. 5.2c)

In the second case (multiple parts identified, but without a temporal ordering of actions), employ the method described in 3a utilizing the *and_seq* decomposition operator in place of *or_seq*. Such a decomposition represents the end user acting on all parts 1–n in any order (not shown in Fig. 5.2). In the third case (one or more specific parts identified, including a temporal ordering of actions:

i. Decompose the sub-activity representing this step into one sub-activity using the *ord* decomposition operator. If there are sub-steps, each sub-activity should be decomposed further into sub-activities by repeating part-3 of this method. Such a decomposition represents the end user acting on parts 1–n in order (n = 1 in Fig. 5.2c). Otherwise, this sub-activity should be decomposed into a corresponding action representing the end user acting on the identified component (encoded generally as "hActOnComponent_identified" in Fig. 5.2f). The analyst should include task execution conditions to identify when activities and actions begins executing, repeat executing, and are complete

After encoding the procedure, the analyst should then instantiate the EOFM input and local variables encoded in activity execution conditions as well as human action variables. Input and human action variables will be utilized to inform the device modeling process (discussed in Section 5.1.2).

To investigate the effects of performing main steps out of order, the analyst can encode a duplicate model that employs the *or_seq* decomposition operator for the top-level activity representing the entire procedure (Fig. 5.2a). Such a technique requires temporal logic specifications that assert a desired ordering of main procedural steps (Fig. 5.2b–d), enabling model checking analyses that can provide positive counterexamples showing an improved ordering. A comparison between two such models is demonstrated in Section 5.3.3.

5.1.2 Device Modeling Technique

Using the technique described in [10], a device model is encoded manually in SAL [68]. Utilizing information about the target system, the initial states for configurable hardware and control systems are identified and encoded within SAL initializations. Variable names are encoded using EOFM

formalism syntax for input variables having the prefix *i*. After encoding initializations, guarded transitions are encoded to control what state(s) the device should assume based on end-user inputs (i.e., human actions) and the device's own algorithms.

In this work, the device modeling technique is augmented to aid the analyst in identifying and quantifying potential completeness problems with respect to procedures in documentation. It proceeds as follows:

1. Based on the target system and precondition(s) encoded in a top-level EOFM-XML activity (e.g. an alarm is engaged), identify the initial system configurations that are possible for this precondition to be satisfied. For example, if one of two different power sources could be in-use when an alarm engages (specified as a top-level activity precondition), either configuration is a possible initial state in the device model. The analyst should encode device model infrastructure enabling either possibility. One way to accomplish this is demonstrated in Section 5.3.5

2. Based on human action variables encoded in the instantiated EOFM-XML representation, the analyst should encode guarded transitions in the formal device model coordinating next-states of the device that emerge when each human action executes

3. Next, the analyst should:

   (a) Quantify the initial configurations that are possible when a procedure begins executing (e.g., if there are two exclusive power sources, two interchangeable cables connecting to either source, and one controller receiving the other end of a cable, there could be as many as four configurations involving one of each component)

   (b) Identify procedural steps addressing an initial configuration and quantify the actions prescribed therein (e.g. step-1 of a procedure instructing the end user to disconnect a particular power source from a particular cable, corresponding to one human action encoded in an instantiated EOFM task model)

   (c) Determine what initial configurations are addressed in prescribed actions

(d) Identify the number of addressed configurations and the number of possible initial configurations (e.g., one of four for a system having four possible initial configurations and instructions addressing one of them). This number serves to quantify completeness of the procedure with respect to the task modeling technique (Section 5.1.1)

5.1.3 System Model Composition

In this work, human-system interaction is abstracted via the exchange of input/output variables in a modular, asynchronous composition of human and device models. The automated translator described in [10] generates model infrastructure ensuring that outputs of the formal task model are recognized in the device model (and vice-versa). Automatically generated syntax representing the system model composition is modified to aid in identifying that the model represents an end user executing a procedure in documentation. It is encoded using the SAL syntax shown below, where the system model is named `documentationProcedure`, the formal task model is named `endUser`, the device model is named `device`, and "`[]`" specifies that the modules are asynchronously composed:

```
documentationProcedure:  MODULE = endUser [] device;
```

5.2 Verification Methodology

As mentioned, procedures in documentation accompanying safety-critical systems should be:

1. Accurate
2. Complete
3. Time efficient

The modeling techniques discussed thus far intend to support the analyst in identifying potential problems with respect to accuracy and completeness. In support of time efficiency, this work leverages linear temporal logic (LTL) specifications and model checking [168].

In formal methods, LTL specifications are commonly encoded to assert desired characteristics of a target system. A symbolic model checker [62] can be invoked to search the model exhaustively for specification violations; and if not violations are detected, the model can be considered safe. In

this work, LTL specifications are encoded to assert undesired characteristics of a procedure with respect to time-efficient ordering of main steps (those encoded as sub-activities in Fig. 5.2b–d). If a model checker detects a violation, a trace through the model returned in a counterexample defines an ordering of steps considered safe with respect to time-efficiency. Otherwise, if the model checker returns *proves*, no violations of the undesired characteristic exist in the model, reflecting a time-efficiency failure. Two such generalizable specifications are provided in this section:

1. One that is intended for time-efficiency analyses of setup, operations, maintenance, and troubleshooting procedures

2. One that is intended for time-efficiency analyses of troubleshooting procedures

Utilizing the task modeling technique described in Section 5.1.1, these specifications can be verified with respect to two models:

1. One representing the end user performing all procedural steps in order as prescribed in accompanying documentation (utilizing the *ord* decomposition operator)

2. One representing the end user performing one or more procedural steps in any order (utilizing the *or_seq* decomposition operator)

For model checking analyses that return *proved* in the first model, the analysis can be repeated in the second model. If allowing the end user to perform steps out of order can result in an improved ordering of main steps, a counterexample will show a trace through the model reflecting a potential time-efficiency improvement.

In any procedure, it could be necessary for a particular preparatory step to be completed successfully such that a later action can be successful. Consider the dialysis machine adverse events described earlier in this chapter: patients were routinely connecting themselves to the machine before properly priming the fluid-filled connection tube [83]. This reflects a potential time-efficiency failure that could emerge if procedural steps are not logically ordered, potentially causing an end user to connect oneself before the tube is properly primed. Successful completion of the preparatory tub-priming

step is necessary to support successful completion of the connection step. The specification in (5.1) exemplifies this case more generally in LTL. It reads, “it is always true (G) that when the end user executes a preparatory action ($hPreparatoryAction$), this implies ($\Rightarrow$) that the completion conditions of an action for which $hPreparatoryAction$ is preparing are satisfied ($CompletionConditions_{hLaterAction}$). This specification could also be encoded to represent completion conditions of many later actions on the right-hand side of $\Rightarrow$, depending on the instantiated model (discussed later in this section).

$$\mathsf{G}(hPreparatoryAction \Rightarrow CompletionConditions_{hLaterAction}) \quad (5.1)$$

Such a specification is called *Preparatory action time inefficiency.* For a model representing the end user executing procedural steps as prescribed in the system’s documentation (i.e., an *ord* model), the model checker returning *proved* would indicate that the steps could be ordered incorrectly. In this case, the analyst should verify the specification again with respect to the *or_seq* model, and a counterexample may show a trace through the model representing a potentially improved ordering of steps.

In regard to troubleshooting procedures, one potentially undesired ordering of steps involves actions that cannot correct the problem executing before ones that can. One way to represent such a problem using LTL is by identifying a corrective action, a non-corrective action, and the completion conditions that result from performing a non-corrective action. Leveraging the formal semantics of EOFM, one such specification is encoded generally in (5.2). It reads, “when a corrective action executes ($hCorrectiveAction$), this implies ($\Rightarrow$) that completion conditions of a non-corrective actions are satisfied ($CompletionConditions_{hNonCorrectiveAction}$). The specification could also be encoded to represent completion conditions of many non-corrective actions on the right-hand side of $\Rightarrow$, depending on the instantiated model (discussed later in this section).

$$\mathsf{G}(hCorrectiveAction \Rightarrow CompletionConditions_{hNonCorrectiveAction}) \quad (5.2)$$

Such a specification is called *Corrective action time inefficiency.* A counterexample to the speci-

fication indicates that troubleshooting steps could be ordered incorrectly. Otherwise, if the model checker returns *proved*, the analyst should attempt verifying the specification with respect to the *or_seq* model. If a counterexample is returned, it could be interpreted as an improved ordering of troubleshooting steps.

As mentioned, these specifications can be encoded to represent completion conditions for many actions. Using the model checking syntax of SAL, this can be accomplished for *Preparatory action time inefficiency* as shown below (explained in the next paragraph]):

```
PreparatoryActionTimeInefficiency:  THEOREM documentationProcedure |-
  LET CompletionConditions_hLaterActions:  BOOLEAN =
    iInputVariable_1 = value_1 AND ... AND iInputVariable_n = value_m IN
      G(hPreparatoryAction => CompletionConditions_hLaterActions);
```

The generic variable `hPreparatoryAction` represents a preparatory action that the analyst has identified in the instantiated model. The statement beginning with `LET` enables the analyst to represent the completion conditions for one or more actions that should execute after the preparatory one. EOFM input variables and value assignments are encoded generally as `iInputVariable_1 = value_1 ... iInputVariable_n = value_m` to represent to identify the completion conditions. The conjunction among them (`AND`) specifies that `CompletionConditions_hLaterActions` is true if all completion conditions are satisfied. If applicable, the analyst could also encode disjunctions among these input variable assignments (demonstrated in Section 5.4.3).

For *Corrective action time inefficiency*, the analyst can encode completion conditions for multiple non-corrective actions using similar SAL syntax:

```
CorrectiveActionTimeInefficiency:  THEOREM documentationProcedure |-
  LET CompletionConditions_hNonCorrectiveActions:  BOOLEAN =
    iInputVariable_1 = value_1 AND ... AND iInputVariable_n = value_m IN
      G(hPreparatoryAction => CompletionConditions_hNonCorrectiveActions);
```

Here, the generic variable `hCorrectiveAction` represents a corrective action that the analyst has identified in the instantiated model, where the procedure should be time-critical (e.g. medical device troubleshooting). Generic EOFM input variables and value assignments (`iInputVariable_1 = value_1 ... iInputVariable_n = value_m`) represent completion conditions for multiple non-corrective actions. As before, the Boolean variable representing completion conditions is true if all

of them are satisfied.

5.3 Case Study: Left Ventricular Assist Device Alarm Troubleshooting

To evaluate applicability of the modeling and verification methodologies, a case study is derived from a left ventricular assist device (LVAD) and a set of troubleshooting instructions from its patient documentation. The analysis entails developing a formal task model of the instructional procedure, a formal device model abstracting human-system interaction, and time-efficiency specifications. The task modeling process is utilized to identify potential accuracy problems with the way tasks involving part–whole components are described, particularly tasks that involve connecting and disconnecting cables. The device modeling process is utilized to aid in identifying potential completeness problems involving initial system configurations that are possible when the alarm engages. To determine what constitutes a time-efficient ordering of steps, actions that may potentially resolve the LVAD alarm (corrective actions) are distinguished from actions that cannot potentially resolve the alarm (non-corrective actions). A safe temporal ordering of troubleshooting steps (encoded as EOFM-XML activities) is defined by instantiating the safety specifications described in Section 5.2. These specifications are verified using the SAL symbolic model checker (SAL-SMC) [68], and model checking results are utilized to identify:

- Potential time-efficiency problems with respect to all main procedural steps performed in order
- Potential time-efficiency improvements with respect to one or more main procedural steps performed in any order

5.3.1 The Device

The LVAD is a mechanical circulatory assist device that is designed to work in conjunction with the native heart. It supplies continuous blood flow support through a single, axial impeller surgically implanted at the apex of the left ventricle. Heart failure patients are typically implanted with an LVAD in order to maintain cardiovascular function until a transplant becomes available or until they have fully recovered from a surgery or cardiac event.

Many of these devices are designed for portability such that external controllers and batteries may be carried using harnesses, holsters and/or straps. Patients must also carry select replacement components at all times in the case of a mechanical malfunction or power supply issues. The device used in this case study may include as many as nine cables, three controllers, three lithium-ion batteries, and two lead reserve batteries.

The case study involves the *pump stopped* alarm procedure. A steady, high pitched audible alarm sounding from a small speaker on the controller alerts the patient that the pump has stopped while the controller's alarm light indicating that the pump has stopped is illuminated. While these alarms are engaged, all external cables, connectors and the battery-in-use may appear to be functioning normally.

The procedure involves the system components in Fig. 5.3. The pump cable (Fig. 5.3f,g) attached to the implanted pump has a two-part connector (Fig. 5.3f) and an output end connecting to the controller. The controller (Fig. 5.3a–c) has two input ports for connecting to the pump (Fig. 5.3a) and a battery respectively (Fig. 5.3c). A 1/2 cell AA battery is stored within a case on the controller (Fig. 5.3b) to power the alarms while a battery is disconnected. A twist-off cap on the case can be loosened to silence an audible alarm and turn off a visual alarm.

Connections between the pump and a controller can be configured in one of two ways:

1. The pump cable can be connected to the abdominal cable while the abdominal cable is connected to a controller

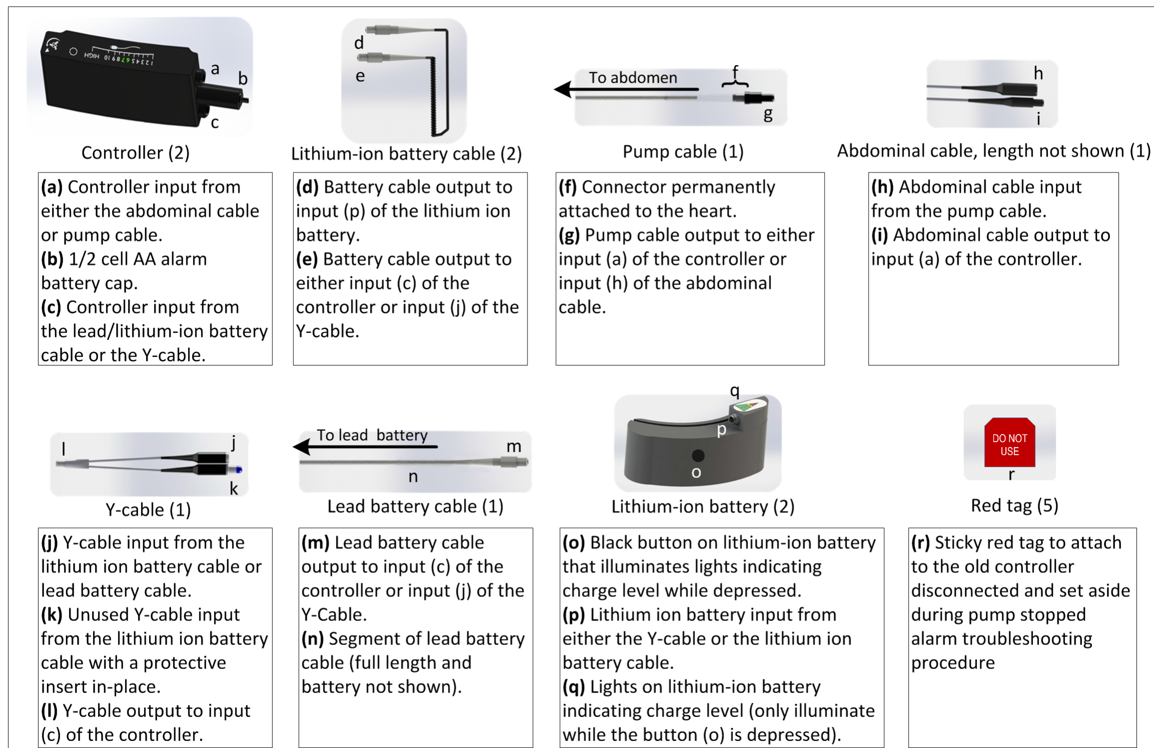
2. The pump cable can be connected directly to a controller, without using an abdominal cable

A connection between the controller and a battery can be configured in one of four ways:

1. One output end of a lithium-ion battery cable is connected to a lithium-ion battery while the other output end is connected to a controller

2. One output end of a lithium-ion battery cable is connected to one input socket of a Y-cable, while the Y-cable output end is connected to a controller

3. The lead battery cable output end is connected to a controller



.

Figure 5.3: Rendering of system components involved in the troubleshooting procedure are listed as "Component Name (Quantity involved in the draft manual procedure and the formal task model)"

4. The lead battery cable output end is connected to one input socket of a Y-cable, while the Y-cable output end is connected to a controller

For the pump to function, one of each pump-to-controller and battery-to-controller connection must be established and the connected battery must have a charge. For an alarm to engage, the in-use controller must have its 1/2 cell AA alarm battery cap tightened.

5.3.2 The Draft Manual

The *pump stopped* troubleshooting procedure in the manual is organized as six ordered steps. The draft instruction manual does not include figures with the component ends labeled as in Fig. 5.3, but they are added to the procedure with caption labels. To aid in understanding the tasks required to complete steps, two of six main steps are listed as multiple, lettered sub-steps (added sub-step letters in italics for 5a, 5b, 6a and 6b):

1. If the pump is stopped:
   (a) Disconnect the abdominal cable *(Fig. 5.3g)* from the pump cable *(Fig. 5.3g)* and set aside all attached components. Disconnect the lithium-ion battery cable *(Fig. 5.3d)* and also partially unscrew the 1/2 cell AA battery cap *(Fig. 5.3b)* on the controller to silence the alarm.
   (b) Plug the pump cable directly into a replacement controller *(Fig. 5.3a)* (eliminating the abdominal cable). Make sure to tighten the 1/2 cell AA battery cap on the replacement controller to activate the alarm. *(Fig. 5.3b)*
   (c) Connect the controller output of a replacement lithium-ion battery cable *(Fig. 5.3e)* to the battery cable input of the replacement controller *(Fig. 5.3c)*. Then connect the other end *(Fig. 5.3d)* to the battery cable input *(Fig. 5.3p)* of a fully charged lithium-ion battery.
2. If the pump is still stopped, call your emergency number immediately.
3. Red tag *(Fig. 5.3r)* all the components of the system that you set aside in step 1a.
4. Make sure that all cables have been changed and then check to see if the connector permanently attached to the heart *(Fig. 5.3f)* is broken. If it is broken and has come apart, put it back together where it is broken. If the heart does not restart, take the connector apart again, rotate the parts 90 degrees, and put it back together again. Repeat three times. The heart may restart.
5. If the heart has still not started, it is possible that you accidentally removed a discharged battery and then plugged the same battery back in by mistake.
   (a) Try changing the batteries again. It is possible that you accidentally removed a discharged lithium-ion battery and then plugged the one back in by mistake.
   (b) If no lights illuminate on either battery while pressing the black button *(Fig. 5.3o)*, disconnect the lithium-ion battery cable from the controller and connect your lead reserve battery .
6. If you have completed all above steps and have carefully replaced all cables and components without successfully restarting the pump:
   (a) Disconnect the power to the heart pump by unplugging whichever battery cable is connected to the replacement controller. There could be a problem with the pump cable that can be repaired without surgery. If you leave the lithium-ion or reserve battery connected, the controller will supply power to the pump, which could be harmful. Disconnecting the battery reduces the chance of a blood clot forming inside the pump by allowing the rotor to spin freely as blood flows across it.
   (b) Partially unscrew the AA battery cap *(Fig. 5.3b)* to silence the alarm.

Figure 5.4: Outline form of case study troubleshooting procedure

5.3.3 Formal Task Model

The six main steps of the troubleshooting procedure are depicted as ten sub-activities of a top-level activity *aRespondToPumpStoppedAlarm*, representing the entire procedure (Fig. 5.5). Individual

steps and sub-steps are depicted in Figs. 5.6–5.14. The top-level activity in Fig. 5.5 has execution conditions specifying that the procedure begins executing when the pump stopped alarm is engaged and completes execution when the alarm disengages (modeling of the alarm status discussed later). Step-1 has three sub-steps (1a–c) that were modeled as main steps because text under the main heading of step-1 ("If the pump is stopped:") does not prescribe an action. Sub-steps of steps 5 and 6 were modeled as main steps for the same reason: because text beside the numbers 5 and 6 in the procedure do not prescribe actions, but sub-steps do, step-5 was modeled as 5a and 5b and step-6 as 6a and 6b. Therefore, the formal task model has ten high-level activities that represent the six main steps: 1a, 1b, 1c, 2, 3, 4, 5a, 5b, 6a, and 6b.

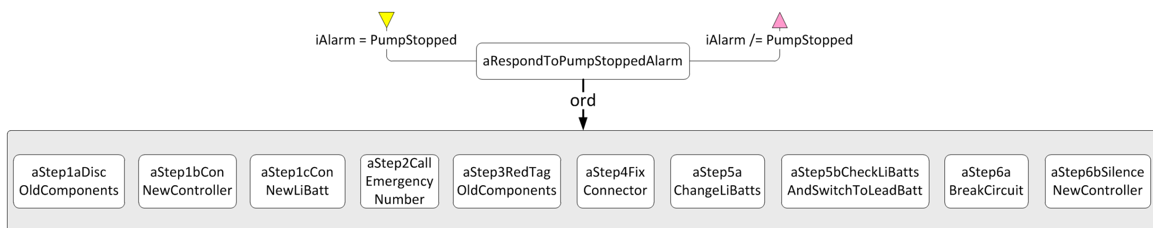


Figure 5.5: Visualization of the six main steps of the troubleshooting procedure encoded as ten EOFM sub-activities. The top-level activity *aRespondToPumpStoppedAlarm* represents the entire procedure, while the ten sub-activities represent end-user activities prescribed within the six main steps. A top-level activity precondition specifies that the procedure begins executing when the pump stopped alarm engages. A completion condition specifies that the procedure completes execution when the alarm disengages. The *ord* decomposition operator specifies that all ten sub-activities must execute in order

Step 1a, *aStep1aDiscOldComponents* appears in Fig. 5.6. It is decomposed by *ord* into four sub-activities representing sub-tasks prescribed in the text:

1. The first sub-activity (*aDiscPumpCableFromAbCable*) represents the first sub-task in the text instructing the end user to disconnect the old abdominal cable from the pump cable. The precondition specifies that the activity begins executing if the pump cable is connected to the old abdominal cable (*iPumpCableToOldAbCable = Connected*). The completion condition specifies that the activity has completed executing when the pump cable is disconnected from the abdominal cable (*iPumpCableToOldAbCable = Disconnected*). Because the task identifies what end of the abdominal cable should be disconnected (the one connected to the pump cable), it is decomposed by *ord* into one human action (*hDiscPumpCableFromAbCable*), which

represents the end user disconnecting the pump cable from the old abdominal cable

2. The second sub-activity (*aSetAsideOldComponents*) represents the second sub-task in the text instructing the end user to set aside old components. The precondition specifies that the activity begins executing if the old components are at-hand (*iOldComponents* = *AtHand*). The completion condition specifies that the activity has completed executing when the old components have been set aside (*iOldComponents* = *SetAside*). No individual components are identified in the task description, and the end user could execute this step in many ways. However, this case study is focused on cable connections, and different ways of setting aside old components are not modeled. This activity is decomposed by *ord* into one human action (*hSetAsideOldComponents*) representing the end user setting aside all old components

3. The third sub-activity (*aDiscOldBattery*) represents the third sub-task in the text instructing the end user to disconnect the lithium ion battery cable. Because there could be two mutually exclusive ways in which the lithium-ion battery cable is initially configured, this activity is decomposed by *xor* into two sub-activities. One sub-activity represents end-user behaviors that are possible if the Y-cable is in-use, while the other represents different behaviors that are possible if the lithium-ion battery cable is connected directly to the controller (i.e., the Y-cable is not in-use). For each sub-activity (either *aDiscOldBattCableFromOldYCable* or *aDiscOld-BattCableFromOldController*), preconditions specify what sub-activity begins executing based on what connections are established: The precondition for *aDiscOldBattCableFromOldYCa-bleInput* specifies that this sub-activity begins executing if the lithium-ion battery cable is connected to the Y-cable (*iOldLiBattCableToOldYCableInput* = *Connected*); the precondition for *aDiscOldBattCableFromOldController* specifies that this sub-activity begins executing if the old lithium-ion battery cable is connected directly to the old controller (*iOldLiBattCable-ToOldController* = *Connected*). Each activity is decomposed further into two sub-activities enabling end-user behaviors that are possible in either case. Because the text does not identify what output end of the lithium-ion battery cable should be disconnected ("disconnect the lithium-ion battery cable"), both sub-activities are decomposed by *or_seq* in order to represent

end-user behaviors that are possible under different preconditions:

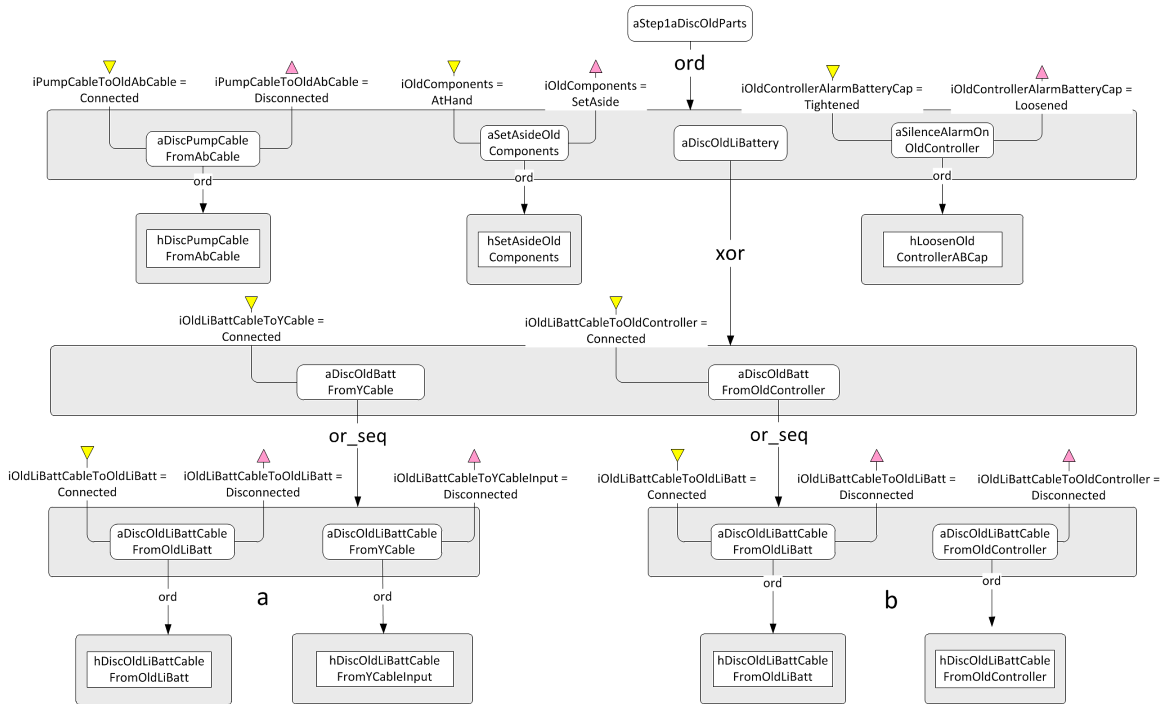


Figure 5.6: Visualization of the formal task model representing step 1a

(a) Sub-activities for *aDiscOldBattCableFromOldYCable* (Fig. 5.6a) represent the end user disconnecting the output end of the lithium-ion battery cable that is connected to the lithium-ion battery (*aDiscOldLiBattCableFromOldLiBatt*), the Y-cable (*aDiscOldLiBattCa-bleFromYCable*), or both:

i. The precondition for *aDiscOldLiBattCableFromOldLiBatt* specifies that this sub-activity begins executing if the old lithium-ion battery cable is connected to the old lithium-ion battery (*iOldLiBattCableToOldLiBatt* = *Connected*). The completion condition specifies that the activity has completed executing when the old lithium-ion battery cable is disconnected from the old lithium-ion battery (*iOldLiBattCable-ToOldLiBatt* = *Disconnected*). The activity is decomposed by *ord* into one human action representing the end user disconnecting the output end connected to the old lithium-ion battery(*hDiscOldLiBattCableFromOldLiBatt*)

ii. The completion condition for *aDiscOldLiBattCableFromYCable* specifies that the ac-

tivity has completed executing when the old lithoum-ion battery cable is disconnected from the Y-cable (*iOldLiBattCableToYCableInput* = *Disconnected*). The activity is decomposed by *ord* into one human action representing the end user disconnecting the old lithium-ion battery from the old Y-cable input (*hDiscOldLiBattCableFromY-Cable*)

(b) Sub-activities for *aDiscOldLiBattCableFromOldController* (Fig. 5.6b) represent the end user disconnecting the old lithium-ion battery cable output end connected to the lithium-ion battery (*aDiscOldLiBattCableFromOldLiBatt*), the old controller (*aDiscOldLiBattCableFromOldController*), or both. Sub-activities are encoded in the same way as *aDiscOld-BattCableFromYCable*, but with sub-activity names, execution conditions, and human actions corresponding to only the old lithium-ion battery cable

4. The fourth sub-activity (*aSilenceAlarmOnOldController*) represents the last sub-task in the text instructing the end user to silence the alarm by loosening the 1/2 cell alarm battery cap on the old controller. The precondition specifies that the activity begins executing if the old controller's alarm battery cap is tightened (*iOldControllerAlarmBatteryCap* = *Tightened*). The completion condition specifies that it is has completed executing when the old controller's alarm battery cap is loosened (*iOldControllerAlarmBatteryCap* = *Loosened*). Because the text identifies the part of the old controller on which the end user should act (the alarm battery cap), it is decomposed by *ord* into one human action representing the end user loosening it (*hLoosenOldControllerABCap*, where "AB" stands for "alarm battery")

Step 1b, *aStep1bConNewController* appears in Fig. 5.7. It is decomposed by *ord* into two sub-activities sub-task prescribed in the text:

1. The first sub-activity (*aConPumpCableToNewController*) represents the first sub-task in the text instructing the end user to connect the pump cable to a new, replacement controller. The precondition specifies that the activity begins executing if the pump cable is disconnected from the old abdominal cable (*iPumpCableToOldAbCable* = *Disconnected*). The completion condition specifies that the activity has completed executing when the pump cable is connected to

the new controller (*iPumpCableToNewController* = *Connected*). Because the pump cable only has one output end, the activity is decomposed by *ord* into one human action representing the end user connecting the pump cable to the new controller (*hConPumpCableToNewController*)

2. The second sub-activity (*aActivateAlarmOnNewController*) represents the second sub-task in the text instructing the end user to activate the alarm by tightening the 1/2 cell alarm battery cap on the new controller. The precondition specifies that the activity begins executing if the new controller's alarm battery cap is loosened (*iNewControllerAlarmBatteryCap* = *Loosened*). The completion condition specifies that it has completed executing when the new controller's alarm battery cap is tightened (*iNewControllerAlarmBatteryCap* = *Tightened*). Because the text identifies the part of the new controller on which the end user should act (the alarm battery cap), the activity is decomposed by *ord* into one human action representing the end user tightening it (*hTightenNewControllerABCap*, where "AB" stands for "alarm battery")

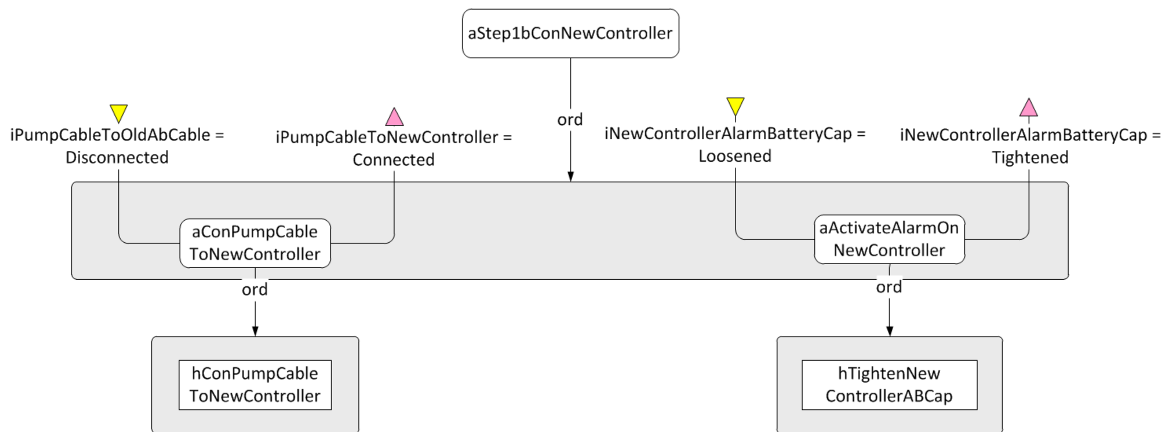


Figure 5.7: Visualization of the formal task model representing step 1b

Step 1c (*aStep1cConFullyChargedLiBatt*) appears in Fig. 5.8. It is decomposed by *ord* into two sub-activities representing each sub-task prescribed in the text:

1. The first sub-activity (*aConNewLiBattCableToNewController*) represents the first sub-task in the text instructing the end user to connect a new lithium-ion battery cable to the new controller. The precondition specifies that the activity begins executing if the new lithium-ion battery cable is disconnected from the new controller (*iNewLiBattCableToNewController*

= *Disconnected*). The completion condition specifies that the activity has completed executing when the new lithium-ion battery cable is connected to the new controller (*iNewLiBattCableToNewController* = *Connected*). Because the text identifies what output end should be connected to the controller, the activity is decomposed by *ord* into one human action representing the end user making the connection (*hConNewLiBattCableToNewController*)

2. The second sub-activity (*aConFullyChargedLiBattToNewLiBattCable*) represents the second sub-task in the text instructing the end user to connect a fully charged lithium-ion battery to the new lithium-ion battery cable. No execution conditions are needed, since they will be specified within its sub-activities (described next). While the task description indicates that only a fully charged battery should be connected, it does not instruct the user to check the charge level first by depressing the black button on either battery (Fig. 5.3o). Therefore, encoding the step as-written requires this sub-activity to be decomposed using the *xor* decomposition operator to specify that the end user could connect either lithium-ion battery (old or new), presuming one or the other is fully charged:

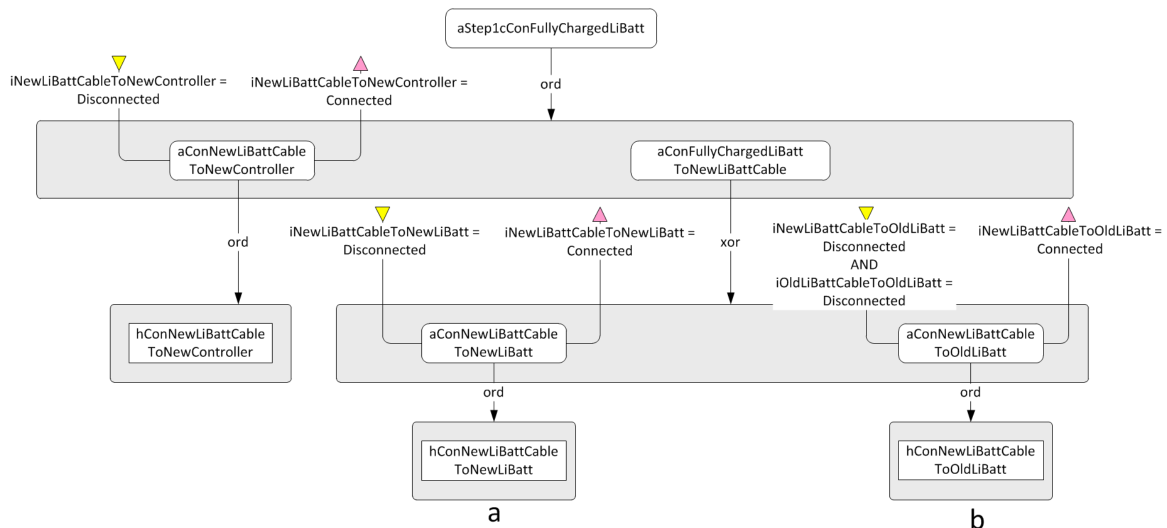


Figure 5.8: Visualization of the formal task model representing step 1c

(a) The first sub-activity (*aConNewLiBattCableToNewLiBatt*, Fig. 5.8a) represents the end user connecting the new lithium-ion battery cable to a new lithium-ion battery that could be fully charged. The precondition specifies that the activity begins executing if

the new lithium-ion battery cable is disconnected from the new lithium-ion battery (*iNewLiBattCableToNewLiBatt = Disconnected*). The completion condition specifies that the activity has completed executing when the new two have been connected (*iNewLiBattCableToNewLiBatt = Connected*). Because the text identifies what output end of the cable should be connected to the battery, the activity is decomposed by *ord* into one human action representing the end user making the connection (*hConNewLiBattCableToNewLiBatt*)

(b) The second sub-activity (*aConNewLiBattCableToOldLiBatt*, Fig. 5.8b) represents the end user connecting the new lithium-ion battery cable to the old, previously disconnected lithium-ion battery. The precondition specifies that this sub-activity begins executing if both the old and new lithium-on battery cables are disconnected from old lithium-ion battery (*iNewLiBattCableToOldLiBatt = Disconnected* `AND` *iOldLiBattCableToOldLiBatt = Disconnected*). The completion condition specifies that the activity has completed executing when the two have been connected (*iNewLiBattCableToOldLiBatt = Connected*). Because the text identifies what output end of the cable should be connected to the battery, the activity is decomposed by *ord* into one human action representing the end user making the connection (*hConNewLiBattToOldLiBatt*) representing the end user making the connection

Step 2 (*aStep2CallEmergencyNumber*) appears in Fig. 5.9a. It is decomposed by *ord* into one human action (*hCallEmergencyNumber*) representing the end user calling an emergency contact number. Step 3 (*aStep3RedTagOldParts*) appears in Fig. 5.9b. No individual components are identified in the task description, and the end user could execute this step in many ways. However, this case study is focused on accuracy with respect to cable connections, so different ways of tagging old components are not modeled. This activity is decomposed by *ord* into one human action (*hRedTagOldComponents*) representing the end user attaching a red tag to all old components.

Step 4 (*aStep4FixBrokenConnector*) appears in Fig. 5.10. The precondition specifies that the activity begins executing if the connector permanently attached to the heart is broken (*iPerma-*

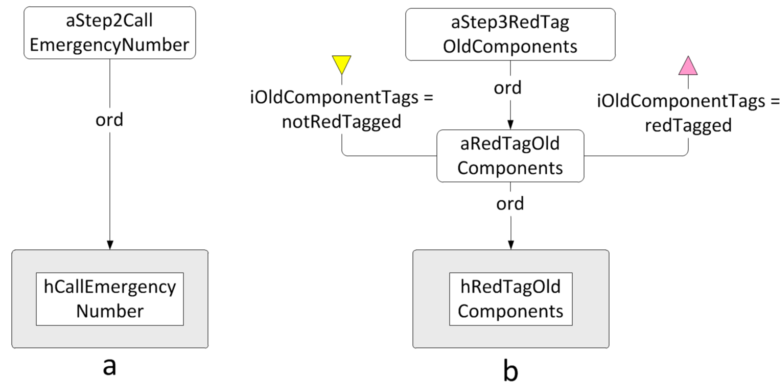


Figure 5.9: Visualization of the formal task model representing steps 2 and 3

nentlyAttachedConnector = *Broken*). Because the text identifies the part on which the end user should act (the broken connector), the activity is decomposed by *ord* into two sub-activities representing each of two sub-tasks prescribed in the text:

1. The first sub-activity (*aReassembleBrokenConnector*) represents the first sub-task identified in the text instructing the end user to reassembled the broken connector. The completion condition specifies that the activity has completed executing when the connector is assembled (*iPermanentlyAttachedConnector* = *Assembled*). Because the text identifies the part on which the user should act ("[the section] where it has come apart"), the activity is decomposed by *ord* into one human action representing the end user reassembling the connector (*hReassembleBrokenConnector*, i.e., reattaching the two connector parts)

2. The second sub-activity (*aTryRotatingParts*) represents the second sub-task identified in the text instructing the end user to try rotating the connector parts three times. The precondition specifies that the action begins executing if the parts have not been rotated (*iRotationCounter* = 0). The repeat condition specifies that the activity repeats executing if the parts have been rotated fewer than three times (*iRotationCounter* < 3). The completion condition specifies that the activity has completed executing when the parts have been rotated three times and the connector has been reassembled (*iRotationCounter* = 0 `AND` *iPermanentlyAttachedConnector* = *Assembled*). Because the text identifies what parts should be acted upon (both connector

parts), the activity is decomposed by *ord* into three human actions. The first action (*hDisassembleConnector*) represents the end user disassembling the connector (i.e., disconnecting the two parts); the second action second action (*hRotateConnectorParts*) represents the end user rotating both connector parts 90°; and the third action represents the end user reassembling the connector (*hReassembleBrokenConnector*)

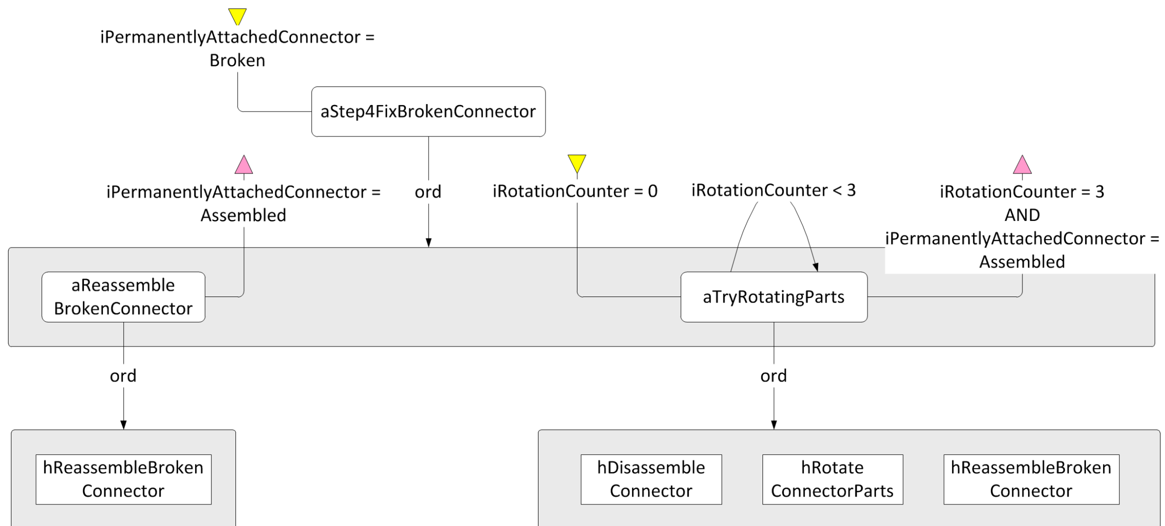


Figure 5.10: Visualization of the formal task model representing step 4

Step 5a (*aStep5aChangeLiBatts*) appears in Fig. 5.11. It is decomposed by *xor* into two sub-activities representing end user behaviors that could execute under different preconditions identified in the text ("Try changing batteries. It is possible that you accidentally removed a discharged lithium-ion battery and then plugged the one back in by mistake"):

1. The first sub-activity (*aSwitchFromOldToNew*) represents an end-user behavior that is possible if the new lithium-ion battery was accidentally connected to the old one earlier in the procedure. The precondition specifies that the activity begins executing if the new lithium-ion battery cable is connected to the old lithium-ion battery (*iNewLiBattCableToOldLiBatt* = *Connected*). The completion condition specifies that the activity has completed executing when the new lithium-ion battery cable has been connected to the new lithium-ion battery (*iNewLiBattCableToNewLiBatt* = *Connected*). The sub-activity is decomposed by *ord* into two human actions representing the end user disconnecting the new lithium-ion battery cable

from the old lithium-ion battery (*hDiscNewLiBattCableFromOldLiBatt*), and then connecting it to the new lithium-ion battery (*hConNewLiBattCableToNewLiBatt*)

2. The second sub-activity (*aSwitchFromNewToOld*) is encoded in the same way as *aSwitchFromOldToNew*, but with execution conditions and human actions modified to represent switching from the new lithium-ion battery to the old one

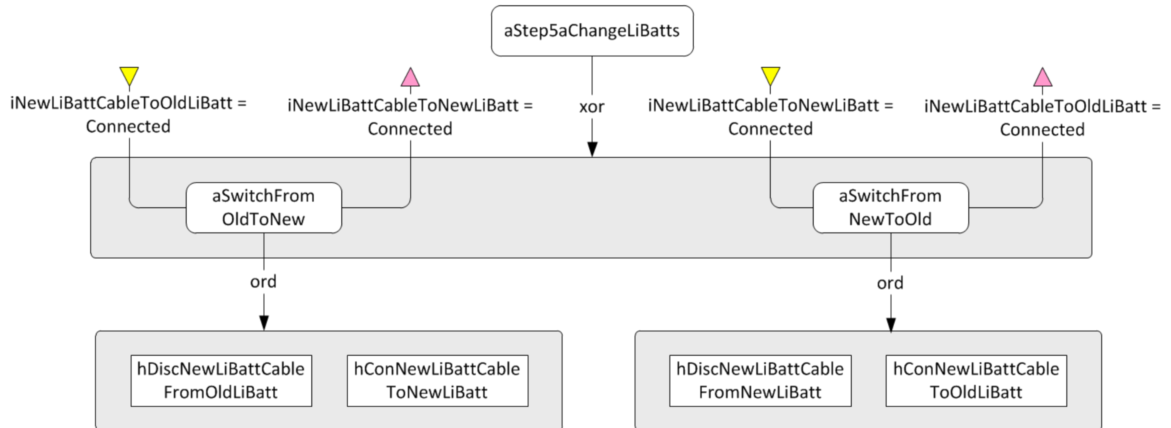


Figure 5.11: Visualization of the formal task model representing step 5a

Step 5b (*aStep5bCheckLiBattsAndSwitchToLeadBatt*) appears in Fig. 5.12. It is decomposed by *ord* into two sub-activities representing each sub-task prescribed in the text:

1. The first sub-activity (*aCheckLiBatteryLevels*) represents the first sub-task identified in the text instructing the end user to check the charge levels on both lithium-ion batteries. Because text identifies both parts involved in the task (the black buttons on both lithium-ion batteries), but no temporal ordering of actions, the activity is decomposed by *and_seq* into two human actions representing the end user checking the charge levels on both lithium-ion batteries (*hDepressBlackButtonOnNewLiBatt* and *hDepressBlackButtonOnOldLiBatt*) in any order

2. The second sub-activity (*aSwitchToLeadOrKeepLiBatt*) represents the second sub-task identified in the text instructing the end user to either keep a lithium-ion battery connected if it is charged or replacing it with the lead reserve battery. The precondition specifies that the activity begins executing if both the old and new lithium-ion batteries are discharged (*iNewLiBatteryLights = 0* AND *iOldLiBatteryLights = 0*). The completion condition specifies that

the activity has completed executing when the lead battery is connected to the new controller (*iLeadBattToNewController* = *Connected*). Because the cable output ends that should be disconnected/connected are identified in text, the activity is decomposed by *ord* into two human actions representing the end user disconnecting the lithium-ion battery cable from the new controller (*hDiscNewLiBattCableFromNewController*), followed by connecting the lead reserve battery to the new controller (*hConLeadBattToNewController*)

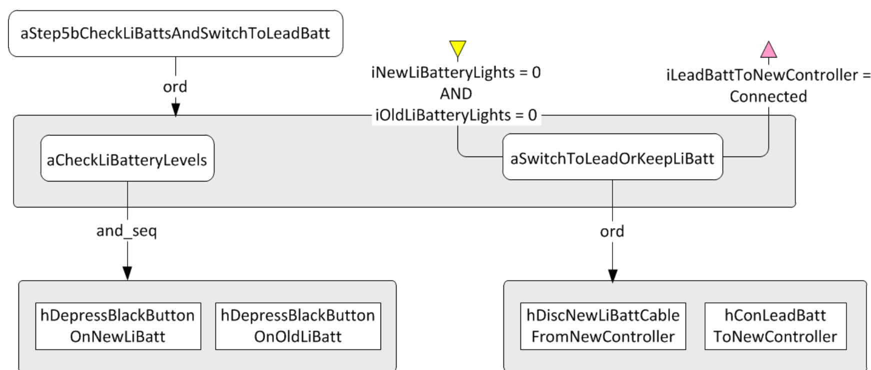


Figure 5.12: Visualization of the formal task model representing step 5b

Step 6a (*aStep6aBreakCircuit*) appears in Fig. 5.13. It is decomposed by *xor* into two sub-activities representing end user behaviors that could execute under different preconditions identified in the text ("[disconnect] whichever battery cable is connected to the controller"):

1. The first sub-activity (*aDiscLeadBatt*) represents an end-user behavior that is possible if the lead battery is connected to the new controllere. The precondition specifies that the activity begins executing if the lead battery is connected to the new controller (*iLeadBattToNewController* = *Connected*). The completion condition specifies that the activity has completed executing when the lead battery is disconnected (*iLeadBattToNewController* = *Disconnected*). The sub-activity is decomposed by *ord* into one human actions representing the end user disconnecting the lead battery from the new controller (*hDiscLeadBattFromNewController*)

2. The second sub-activity (*aDiscLiBatt*) is encoded in the same way for the new lithium-ion battery cable

Step 6b (*aStep6bSilenceNewController*) appears in Fig. 5.14. The precondition specifies that the

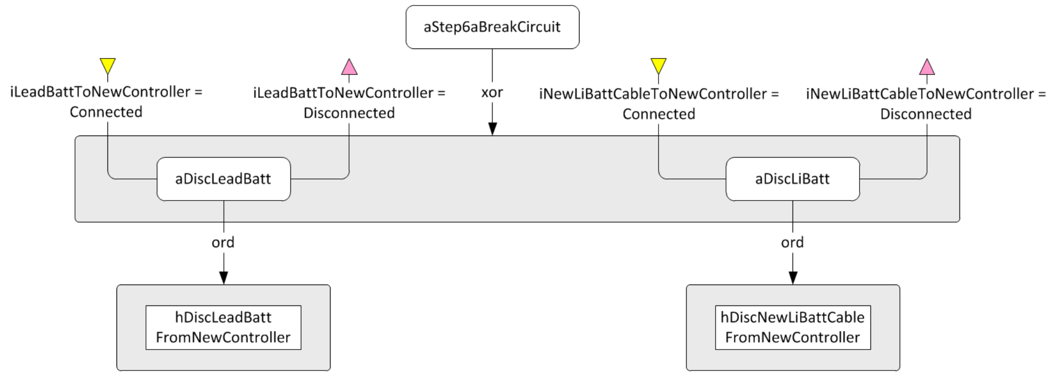


Figure 5.13: Visualization of the formal task model representing step 6a

activity begins executing if the new controller's alarm battery cap is tightened (*iNewController-AlarmBatteryCap* = *Tightened*). The completion condition specifies that it is has completed executing when the battery cap is loosened (*iNewControllerAlarmBatteryCap* = *Loosened*). Because the text identifies the part on which the end user should act (the battery cap), it is decomposed by *ord* into one human action representing the end user loosening it (*hLoosenOldControllerABCap*, where "AB" stands for "alarm battery")

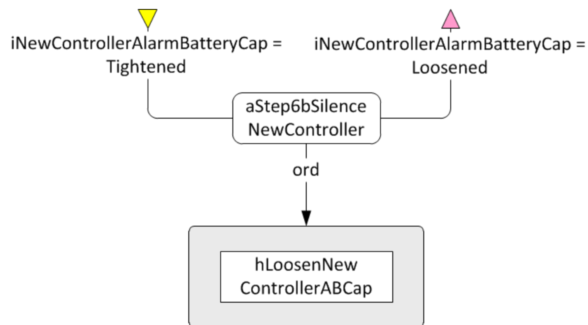


Figure 5.14: Visualization of the formal task model representing step 6b

5.3.4 Translating from XML to SAL

Two versions of the 284-line EOFM-XML representations (Appendix E.1) were translated into SAL using the translator described in [10]. Both models were 754 lines of SAL code.

5.3.5 Formal Device Model

A human-device-interaction (HDI) model was assembled manually using SAL [68]. This HDI model is a formal representation of initial device component states that are possible when the pump stopped alarm engages, including cable connections, alarms engaged on the controller in-use, alarm battery caps on controllers, red-tagged old components, lights illuminated on batteries, the connector permanently attached to the heart, the number of 90° rotations performed on the connector, and the position of old components relative to the patient. The LVAD components shown in Fig. 5.3 and the input variables encoded in the formal task model were used to identify one such set of configurations considered functional; i.e., the controller, pump, cables, and one power source are connected in a way that supports normal pump operation.

One set of initializations concerns connections that are possible between the controller and pump, which may or may not involve the abdominal cable. These configurations and the corresponding SAL syntax for encoding them are listed below.

1. The pump cable output end may be connected to or disconnected from the input socket of the abdominal cable:

```
iPumpCableToOldAbCable IN {Connected, Disconnected};
```

2. If the pump cable output end is connected to the abdominal cable input port, then it is disconnected from the old controller input port:

```
iPumpCableToOldController =
  IF iPumpCableToOldAbCable = Connected
    THEN Disconnected
    ELSE Connected
  ENDIF;
```

3. If the pump cable output end is connected to the old controller input socket, then the abdominal cable is disconnected from the old controller:

```
iAbCableToOldController =
  IF iPumpCableToOldController = Connected
    THEN Disconnected
    ELSE Connected
  ENDIF;
```

A second set of initializations concerns connections that are possible between the controller and a

power supply, which may or may not involve the Y-cable. These configurations and the corresponding SAL syntax for encoding them are listed below.

1. The Y-cable output end may be connected to or disconnected from the input socket of the old controller:

```
iYCableToOldController IN {Connected, Disconnected};
```

2. If the Y-cable output end is connected to the old controller, then the old lithium-ion battery cable is disconnected from the old controller. Otherwise, the old lithium-ion battery cable may be connected or disconnected from the old controller:

```
iOldLiBattCableToOldController IN
  IF iYCableToOldController = Connected
    THEN {Disconnected}
  ELSE {Connected, Disconnected}
  ENDIF;
```

3. If a Y-cable or lithium-ion battery cable output end is connected to the old controller input socket, then the lead battery cable output end is disconnected from the old controller. Otherwise, the lead battery cable output end is connected to the old controller:

```
iLeadBattToOldController =
  IF iYCableToOldController = Connected
    THEN Disconnected
  ELSIF iOldLiBattCableToOldController = Connected
    THEN Disconnected
  ELSE Connected
  ENDIF;
```

4. If the Y-cable output end is connected to the old controller input socket, then the old lithium-ion battery cable is either connected to or disconnected from the Y-cable. Otherwise, it is disconnected:

```
iOldLiBattCableToOldController IN
  IF iYCableToOldController = Connected
    THEN {Connected, Disconnected}
  ELSE {Disconnected}
  ENDIF;
```

5. If an old lithium-ion battery cable output end is disconnected from the Y-cable input socket and the Y-cable output end is connected to the old controller input socket, then the lead battery cable is connected to the Y-cable. Otherwise, it is disconnected:

```
iLeadBattToYCable =
  IF (iOldLiBattCableToYCable = Disconnected AND
      iYCableToOldController  = Connected
    THEN Connected
  ELSE Disconnected
  ENDIF;
```

6. If an old lithium-ion battery cable output end is connected to the old controller input socket or

   (a) The Y-cable output end is connected to the old controller input socket and

   (b) The lead battery cable output is disconnected from the Y-cable input socket

   then the old lithium-ion battery's other output end is connected to the old lithium-ion battery input socket. Otherwise, it is disconnected:

```
iOldLiBattCableToOldLiBatt =
  IF iOldLiBattCableToOldController = Connected OR
    (iYCableToOldController  = Connected AND
     iLeadBattToYCable       = Disconnected)
    THEN Connected
  ELSE Disconnected
  ENDIF;
```

Remaining initializations concern variables that do not represent input–output cable connections that were established when the alarm engaged:

- The alarm, which is initialized as *PumpStopped*:

```
iAlarm = PumpStopped;
```

- The placement of old components relative to the user, which are at hand when the alarm engages:

```
iOldComponents = AtHand;
```

- The connector permanently attached to the heart, which may be broken or assembled when the alarm engages:

```
iPermanentlyAttachedConnector IN {Broken, Assembled};
```

- The number of times broken connector parts have been rotated 90°, which is 0 when the alarm engages:

```
iRotationCounter = 0;
```

- "Do not use" red tag attached to old components, which are all not attached when the alarm engages:

```
iOldComponentTags = notRedTagged;
```

- The number of charge indicator lights on lithium-ion batteries when the alarm engages, which are both 0. The lead battery light is 0 if it is disconnected from the old controller; otherwise, if it is connected, it could have one light indicating that is has a charge or no light indicating that it is discharged.

```
iOldLiBatteryLights = 0 ;
iNewLiBatteryLights = 0;
iLeadBatteryLight IN
  IF iLeadBattToOldController = Connected
    THEN {0, 1}
  ELSE {0}
  ENDIF;
```

- Alarm battery caps on the controllers, which are tightened and loosened respectively for the old and new controllers when the alarm engages:

```
iNewControllerAlarmBatteryCap = Loosened;
iOldControllerAlarmBatteryCap = Tightened;
```

- Cable connections involving new, replacement components, which are all disconnected when the alarm engages:

```
iNewLiBattCableToNewLiBatt     = Disconnected;
iNewLiBattCableToNewController = Disconnected;
iLeadBattToNewController       = Disconnected;
iPumpCableToNewController      = Disconnected;
```

Guarded transitions were encoded for all human action variables represented within the instantiated EOFM-XML representation of the troubleshooting procedure. These transition statements are depicted graphically in Figures 5.15 and 5.16. Transition statements were encoded individually for each component; however, Fig. 5.15a represents all 14 state transitions for cable connections and disconnections, which are all encoded in a similar way. Fig. 5.15b and Fig. 5.15c represent state transitions for the alarm battery cap on the old and new controllers respectively. Fig 5.16 represents state transitions for all other EOFM input variables. Because the case study is concerned with modeling all steps of the troubleshooting procedure, the alarm state does not transition.

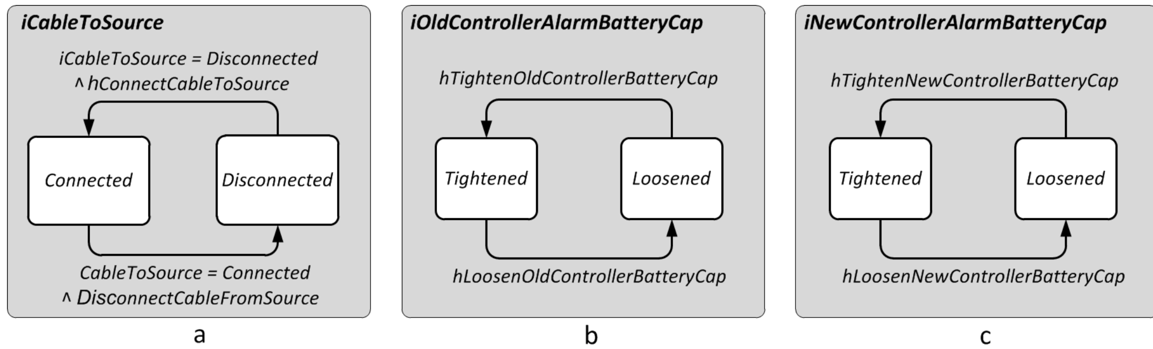


Figure 5.15: Visual representations of device state transitions encoded in the HDI model. Text written directly above and below arrows define the conditions that must evaluate to *true* for the respective state transition to execute. (a) Cable connections and disconnections. (b) State of the alarm battery cap on the old controller. (c) State of the alarm battery cap on the new controller

The input variables *iOldLiBatteryLights* (Fig. 5.16e) and *iNewLiBatteryLights* (Fig. 5.16f) represent lithium-ion battery charge levels. As described in Fig. 5.3, the lights on either lithium-ion battery may only illuminate while the black button (Fig. 5.3o) is depressed. However, the transitions encoded in the formal device model represent a permanent change-of-state for *iOldLiBatteryLights* and *iNewLiBatteryLights* that take effect after execution of *hDepressBlackButtonOnNewLiBattery* and *hDepressBlackButtonOnOldLiBattery* respectively. The lead reserve battery does not have this button, and the light indicating that it has a charge can transition permanently to 0 (indicating that it is discharged) or 1 (indicating that it is not discharged) after it is connected to a controller (Fig. 5.15g).

A final guarded transition was encoded at the end of the device model to remove deadlock states, or states in which no transition guards are satisfied:

```
[]ELSE -->
  Ready’ = IF (Ready AND Submitted)
    THEN FALSE
    ELSE Ready
  ENDIF;
```

The final human-device interaction model was 258 lines of SAL code (Appendix E.2). As mentioned, human-device interaction is abstracted via the asynchronous composition of task and HDI models: transitions in the task model occur first, and if any human action executes, its effects on

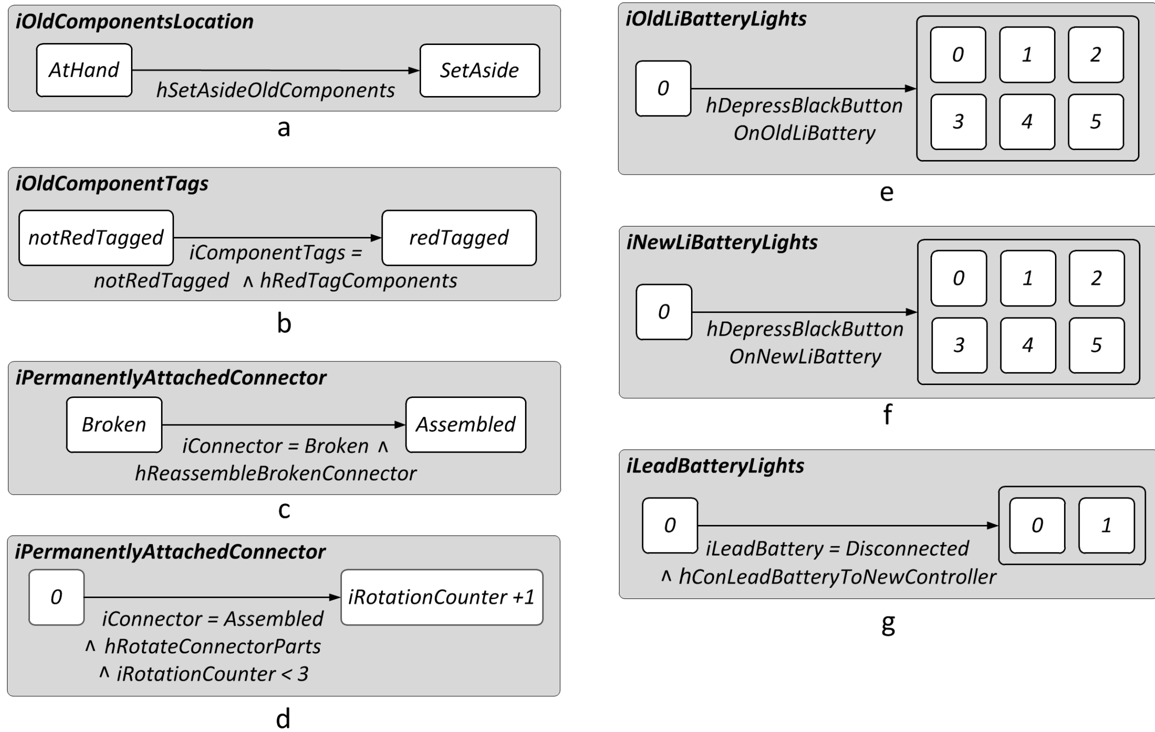


Figure 5.16: Visual representations of device state transitions encoded in the HDI model. Text written directly below arrows states the conditions that must evaluate to *true* for the respective state transition to execute. Variable names are listed in bold italic text within each gray, rounded-edge rectangle. (a) Position of all old, removed components relative to the patient. (b) Attaching red tags (Fig.5.3r) to old, removed components. (c) Status of the connector permanently attached to the heart (Fig. 5.3f). (d) 90° rotations performed on the connector permanently attached to the heart (Fig. 5.3f). (e, f, g) Lights illuminated on the old lithium-ion battery (Fig. 5.3q), new lithium-ion battery (Fig. 5.3q), and lead reserve battery respectively. Each of these variable values can transition to any one of the numbers within white, rounded edge squares

the device are observed in the next-state. The case study model composition is encoded in SAL as shown in Section 5.1.3.

5.3.6 Verification

Two LTL specifications were encoded to assert time-efficiency properties of the procedure. One *Corrective action time inefficiency* specification asserts that the patient will label the old components with red tags before fixing the connector permanently attached to the heart. This situation is not time efficient because attaching a red tag to old components cannot potentially restart the pump (i.e., it is a non-corrective action). Fixing the connector permanently attached to the heart, however,

is a potentially corrective action. The specification is encoded in SAL as shown below, where `hReassembleBrokenConnector` is the corrective action and `iOldComponentTags = redTagged` is the completion condition of the non-corrective action. The specification reads, "it is always true that when the end user reassembles the broken connector permanently attached to the heart, the old components have been red tagged."

```
G(hReassembleBrokenConnector => iOldComponentTags = redTagged);
```

One *Preparatory action time inefficiency* specification asserts that patient will connect either an old or new lithium-ion battery to the new lithium-ion battery cable before checking the new lithium-ion battery level. This situation is not time efficient because checking a battery's charge level before connecting it constitutes a preparatory action; failing to do so could result in the end user connecting a discharged battery, which cannot restart the pump. The specification is encoded in SAL as shown below. The `LET` statement specifies the completion conditions of multiple actions that should execute after the preparatory actions. They include:

- The new lithium-ion battery cable is connected to the new controller (`iNewLiBattCableToNewController = Connected`), which is a completion condition for an activity having the action `hConNewLiBattCableToNewController`, and:
  - The new lithium-ion battery cable is connected to the new lithium-ion battery (`iNewLiBattCableToNewLiBatt = Connected`), which is a completion condition for an activity having the action `hConNewLiBattCableToNewLiBatt`, or
  - The new lithium-ion battery cable is connected to the old lithium-ion battery (`iNewLiBattCableToOldLiBatt = Connected`), which is a completion condition for an activity having the action `hConNewLiBattCableToOldLiBatt`

The specification reads, "it is always true that when the end user checks the charge level of the new lithium-ion batter, the completion conditions of actions that should execute later are satisfied (i.e., one of the batteries is powering the new controller)."

```
PreparatoryActionTimeInefficiency:  THEOREM documentationProcedure |-
  LET CompletetionCondition_hLaterActions:  BOOLEAN =
    iNewLiBattCableToNewController = Connected AND
      (iNewLiBattCableToNewLiBatt  = Connected OR
       iNewLiBattCableToOldLiBatt  = Connected) IN
       G(hPreparatoryAction => CompletetionConditio_hLaterActions);
```

The model checking analyses were conducted twice: once for the *ord* task model; i.e., the activities must execute in the order they are written, and once for the *or_seq* task model. All verifications were done using SAL-SMC [68]. Verification reports for counterexamples were visualized using the graphical notation described in [73].

5.4 Results

All three steps of the methodology produced results:

1. Encoding the written procedure in EOFM-XML revealed a potential accuracy problem (discussed in Section 5.4.1)

2. Encoding the formal device model manually in SAL revealed a potential completeness problem (discussed in Section 5.4.2)

3. Verifying *Preparatory action time inefficiency* and *Corrective action time inefficiency* specifications revealed two potential time-efficiency problems (discussed in Section 5.4.3)

5.4.1 Encoding of Written Procedure in EOFM-XML

While applying the task modeling technique (Section 5.1.1), a potential accuracy problem was uncovered in the written troubleshooting procedure: the instructions for step-1a state, "disconnect the lithium-ion battery cable"; however, the output end that should be disconnected is not identified. Representing this step formally required decomposing the sub-activity for step-1a using the *or_seq* operator. The resulting decomposition reflects three possible end-user actions that could execute in any order:

1. Disconnecting the lithium-ion battery cable from the battery

2. Disconnecting the lithium-ion battery cable from the controller

3. Disconnecting the lithium-ion battery from both the battery and the controller

Consequences of this potential accuracy problem became apparent while encoding preconditions for step-1c, which instructs the end user to connect a fully charged lithium-ion battery to the new battery cable: if the old lithium-ion battery is fully charged, the end user could connect it to the new lithium-ion battery cable; alternatively, if the new lithium-ion battery is fully charged, the end user could connect this one instead. Exactly one battery can be connected in this step (specified using the *xor* decomposition operator), and either one could be connected if there are five charge indicator lights illuminated on either battery (indicating that it is fully charged, specified within activity preconditions). However, the task description did not specify that the end user should check a battery's charge level before connecting it. Additionally, if a malfunction of the old lithium-ion battery originally caused the pump stopped alarm to engage, reconnecting this battery in step-1c cannot restart the pump, regardless of its charge level. Thus, by disconnecting the lithium-ion battery cable from the battery cable in step-1a, it becomes possible for the end user to connect a discharged or malfunctioning battery to the new lithium-ion battery cable in step-1c. This reflects an accuracy problem encoded in the model (and potentially emergent for an end user of the LVAD).

5.4.2 Encoding The Device Model

While applying the device modeling technique (Section 5.1.2), it was observed that all initial cable, battery, and controller configurations were not addressed in the written instructions, reflecting a potential completeness problem. As mentioned in Section 5.3.1, one of two initial pump-to-controller and one of four initial battery-to-controller connections make up possible eight functional configurations of the case study device, and the pump stopped alarm could engage when one such configuration is established. All eight were encoded within initializations of the formal device model. One aspect of completeness for this procedure requires that cable-disconnection actions in steps 1a and 1b (disconnecting old components) address all initial configurations. Three such actions were prescribed in these steps and then encoded within guarded transitions of the formal device model:

1. `hDiscPumpCableFromAbCable`

2. `hDiscOldLiBattCableFromOldController`

3. `hDiscOldLiBattCableFromYCable`

However, these actions were insufficient for addressing all eight configurations, reflecting a potential completeness problem (Table 5.2).

Table 5.2: Results of encoding the formal device model

Initial configuration (as list of components)	Applicable actions encoded in the formal device model	Configuration fully addressed

• Pump cable
• Abdominal cable
• Controller
• Y-cable
• Lithium-ion battery cable
• Lithium-ion battery
	`hDiscPumpCableFromAbCable`
	`hDiscOldLiBattCableFromOldController`
	`hDiscOldLiBattCableFromYCable`
		✓

• Pump cable • Abdominal cable • Controller • Lithium-ion battery cable • Lithium-ion battery	`hDiscPumpCableFromAbCable` `hDiscOldLiBattCableFromOldController`	✓
• Pump cable • Abdominal cable • Controller • Y-cable • Lead battery	`hDiscPumpCableFromAbCable`	✗
• Pump cable • Abdominal cable • Controller • Lead battery	`hDiscPumpCableFromAbCable`	✗
• Pump cable • Controller • Y-cable • Lithium-ion battery cable • Lithium-ion battery	`hDiscOldLiBattCableFromOldController` `hDiscOldLiBattCableFromYCable`	✗
• Pump cable • Controller • Lithium-ion battery cable • Lithium-ion battery	`hDiscOldLiBattCableFromOldController`	✗
• Pump cable • Controller • Y-cable • Lead battery		✗
• Pump cable • Controller • Lead battery		✗

This result indicates that two of eight initial configurations are addressed in the procedure.

5.4.3 Formal Verification of Time-Efficiency Specifications

Formal verification results are presented in Table 5.3. The model checker returned "proved" for both specifications in the *ord* model, indicating that time-efficiency problems could emerge for the end user when the six main procedural steps (encoded as ten heterarchical EOFM activities) are executed in order. Counterexamples were returned for both specifications in the *or_seq* model, reflecting potentially improved orderings or procedural steps with respect to one preparatory action and one corrective action.

Table 5.3: Case study model checking results

Specification name	Task model					
	ord			*or_seq*		
	Result	States visited	Time (s)	Result	States visited	Time (s)
Preparatory action time inefficiency	proved	26,760	5.54	counter-example	1,680	3.69
Corrective action time inefficiency	proved	26,760	5.58	counter-example	1,680	3.62

The *or_seq* model counterexample to *Preparatory action time inefficiency* is shown in Fig. 5.17. A two-step trace shows the end user depressing the black button on the new lithium-ion battery to check its charge level before connecting either lithium-ion battery to the new lithium-ion battery cable. Such a task ordering could be improved with respect to time-efficiency: checking the new lithium ion battery's charge level before connecting it could potentially prevent the end user from connecting a discharged battery.

The *or_seq* model counterexample to *Corrective action time inefficiency* is shown Fig. 5.18. A two-step trace shows the end user reassembling the broken connector that is permanently attached to the heart (i.e., executing a corrective action) before attaching a red tag to the old controller (i.e., completion conditions of a non-corrective action are not satisfied). Such a task ordering could be improved with respect to time efficiency, since attempting to fix the broken connector can potentially resolve the alarm, while attaching red tags to previously disconnected components cannot.

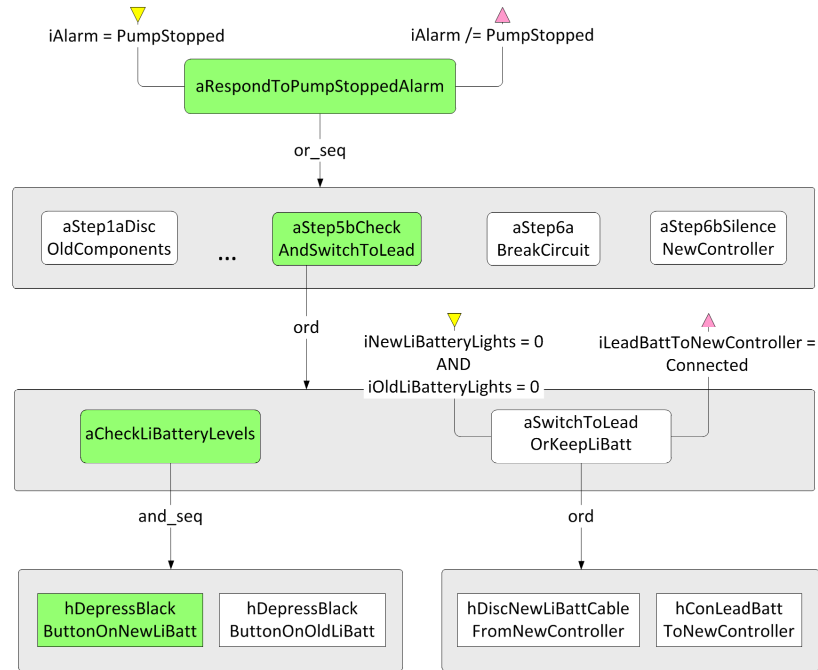


Figure 5.17: Visualization of the two-step counterexample to *Preparatory action time inefficiency* in the *or_seq* model. Green, rounded-edge rectangles are activities that are executing. The green, square-edge rectangle indicates that *hReassembleBrokenConnector* is valued true in the state violating the specification

5.5 Discussion

This chapter has presented a novel application of formal methods for representing and evaluating written procedures accompanying a human-interactive system. In support of documentation procedure accuracy, the task modeling technique intends to support the analyst in identifying potential accuracy problems while attempting to represent an end user executing procedural steps as-written. Leveraging the formal semantics of EOFM [10], the method provides a way of modeling different combinations of end-user actions that are derived from what components/parts are identified in the user manual. In support of completeness, the device modeling technique aids in comparing initial system configurations that are possible when a procedure begins executing with configurations addressed by actions in the procedure. Initial configurations and actions addressing them are quantified to identify potential completeness problems. The ratio of addressed-to-initial configurations is expressed as a fraction to aid in quantifying one kind of documentation procedure completeness

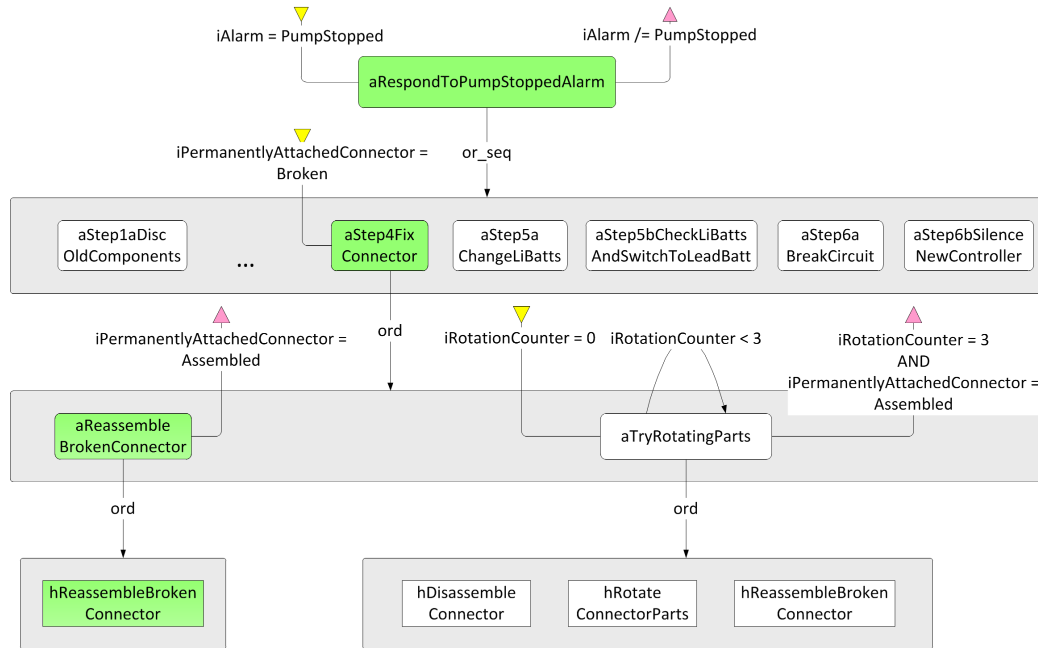


Figure 5.18: Visualization of the two-step counterexample to *Preparatory action time inefficiency* in the *or_seq* model. Green, rounded-edge rectangles are activities that are executing. The green, square-edge rectangle indicates that *hReassembleBrokenConnector* is valued true in the state violating the specification

problem. In support of time efficiency, a verification methodology provides LTL specifications for asserting undesired temporal orderings of main procedural steps with respect to:

- Completion conditions that are satisfied for one or more actions that should come after a necessary preparatory action (with general application to safety-critical system procedures)
- Completion conditions that are satisfied for one or more non-corrective actions that should come after a corrective action (with particular application to troubleshooting procedures)

Specifications can be encoded and verified using symbolic model checking, and results for two models can be compared:

1. One model representing the end user executing all main procedural steps in order
2. One model representing the end user executing one or more main procedural steps in any order

The approach was applied in a case study based on a medical device troubleshooting procedure. The task modeling technique helped identify a potential accuracy problem in a procedural

step, which while the device modeling technique revealed that the procedure does not prescribe cable disconnection actions addressing all initial configurations. One set of model checking analyses "proved" that the procedure could have time-efficiency problems involving incorrect orderings of one preparatory and one corrective action. A second set of model checking analyses aided in identifying potentially improved orderings of procedural steps with respect to the specifications. These results indicate that each step of the approach could be useful for improving accuracy, completeness, and time efficiency respectively:

1. Encoding the formal task model was useful for identifying a potential accuracy problem
2. Encoding the formal device model was useful for identifying a potential completeness problem
3. Formally verifying LTL specifications and comparing results in two models was useful for identifying potential time-efficiency problems as well as potential improvements

5.5.1 Methodological Considerations of the Modeling Approach

Leveraging the EOFM-XML language and translation tool [10] yielded a new modeling technique for representing an end user executing a procedure as it is written in a target system's accompanying documentation. While this work appears to be the first to leverage formal task modeling in this way, other researchers have represented user documentation programmatically [169], [170]. Human-computer interaction (HCI) researchers have also leveraged formal methods for user documentation generation [171], [172] and documentation evaluation by formal specification [173], [174]. Other methods [175], [176] have demonstrated that a careful reading of the device's documentation as part of a formal device usability evaluation or interface design may reveal errors in the documentation. Reading user documentation with the intention of using it to develop a formal representation of the device has similar value to formally encoding written task descriptions in the documentation. Either method may expose potential usability problems while paying particularly close attention to formal device descriptions (or lack thereof) within the procedure. However, in this work both methods together (representing the procedure and device formally) proved complementary. In regard to accuracy:

- Device descriptions in the user manual helped to identify the part-whole composition of device components
- Device descriptions in the procedure helped to identify potential end-user actions that could execute based on what part-whole components are identified

In regard to completeness:

- Device descriptions in the user manual helped identify to initial cable, controller, and battery configurations that are possible when the pump stopped alarm engages
- The set of human action variables encoded in the task model helped identify guarded transitions in the formal device model
- Comparing the number of possible initial configurations (8) with those addressed in steps 1a and 1b of the procedure aided in:
  - Identifying a potential completeness problem (not all initial configurations are addressed in the procedure)
  - Quantifying the potential completeness problem (2 of eight initial configurations addressed in steps 1a and 1b)

Together, the task and device modeling techniques could be complementary with respect to accuracy and completeness.

5.5.2 Methodological Considerations of the Verification Approach

Specifications comprise a major feature of this analysis, and there are several advantages to encoding them using LTL. In formal methods, a safety specification describes a feature of code in which some undesirable state is never reached [177]. In the documentation domain, LTL allows us to express these descriptions as instances of undesired conditions so potential improvements could be identified in counterexamples. However, there are several considerable challenges to using LTL effectively: One is that these propositions can be difficult to develop and contextualize. This problem has been

a topic in formal methods for nearly two decades [107], [178] and it has been addressed in the context of formally verifying human-automation interaction [179].

Another problem with developing safety specifications is vacuous truth, which results in a specification being spuriously proven by a model checker. Vacuous truth occurs when an analyst encodes a safety specification using the implication operator ($\Rightarrow$), but the Boolean function on the left-hand side of the operator never evaluates to *true* in the system model. This was avoided in the current work by developing relatively simple specifications, but they can be avoided in more complex specification by developing lemmas that verify the existence of state(s) defined on the left-hand side of the implication operator.

In this research, specifications were based on practical definitions of corrective actions, non-corrective actions, preparatory actions, action completion conditions, and time-efficient temporal relationships among them. However, while this definition of time efficiency can be verified using symbolic model checking, the method did not prove that this definition is indeed medically valid.

5.5.3 Future Work

Formal task and device models developed in this research need to be encoded manually. Some formal methods-based frameworks such as IVY [59] and ADEPT [48] have graphical development environments for representing human-interaction device; while task modeling tools such as Little-JIL [132] and ConcurTaskTrees [66] have similar environments for representing human task behavior. Future work should explore ways of integrating these graphical tools to facilitate the formal modeling of:

- An end user executing a procedure as-written
- Initial hardware configurations
- Configurations that are addressed by end-user actions prescribed in the procedure

Currently, model checking software such as SAL-SMC can only handle a limited number of variables and transitions. If the analyst encodes an EOFM task model with too much detail or with too many states, then the model checker will not successfully produce a verification report

and it will instead indicate that it has run out of memory. To leverage a model checking tool like SAL-SMC, analysts must encode a formal task model that accurately represents a written procedure at a complexity level commensurate with the computational capability of symbolic model checking. Computational complexity may also limit the scalability of the modeling technique to EOFMs that accurately represent longer written procedures or an entire document. Methods exist for the decreasing the state space of symbolic, finite state model (see for example [180]), and they can be applied to formally verify safety properties of written procedures with more states and state transitions (and perhaps entire documentation). Other tools included with the SAL package such as bounded model checking (BMC) and witness model checking (WMC) can also be used to evaluate formal task analytic models of documentation procedures. Future work should investigate the scalability of this method as well as compare different model checking technologies for evaluating documentation.

In regard to time-efficiency specifications, future work should incorporate a method for ensuring their validity with respect to the target system. One possible avenue for achieving this goal is leveraging specifications from unsafe conditions identified during failure mode and effects analysis (FMEA) [181] or other risk analysis methods commonly used during early stages of safety-critical system design.

Chapter 6: A Formal Approach to Hardware Configurability: Modeling, Specification, and Verification of Gibsonian Affordance

As shown in Chapter 5, it is critical that instructional procedures are written in a way that supports end users in completing them correctly. However, even if procedures are well written, there is no guarantee that an end user will follow them in every situation or execute every task exactly as prescribed. Additionally, there are times when certain physical manipulations of interface components should be avoided. One way interface designers address this is by designing configurable hardware components with characteristics that enable safe opportunities for action (called affordances) at the appropriate times, while disabling unsafe ones at all times.

The psychologist J.J. Gibson defined affordance as "what an environment offers the animal, what it provides or furnishes, either for good or ill," [82] where an environment is a three-dimensional space and all entities therein, including the animal. Since this work applies affordance toward HFE, the term *human* is used in place of *animal*.

Gibson's definition identifies the existence of an human-environment system (HES), where affordances are directly perceivable properties of the HES that "point both ways" [82] (i.e. to human and environment) so that human and environment are complements with respect to affordance, and neither can be considered separately. One example of affordance is door openability for a door with a turning knob, which could be determined by considering the position of a door with respect to a human, the size and position of the door knob, and two human motor capabilities: applying enough force to turn the knob and pulling it hard enough to open the door. These motor capabilities must exist regardless of where the human operator is standing, and the affordance of door openability emerges when the spatial positions of the door and human operator allow it (i.e. when relations among entities in the environment and human motor capabilities co-occur).

An interpretation of affordance has been used to inform design in human-computer interaction (HCI) because it is useful for reasoning about the cognitive and perceptual functions involved in

human behavior. Gibson's definition of affordance has thus been modified in HCI (sometimes called "perceived affordances" or "signifiers," discussed further in Chapter 7) by incorporating human cognition [182]. Gibson, however, asserts that affordances are directly perceivable in a way that relaxes assumptions about how human cognitive functions influence their existence [82]. Gibsonian affordance is therefore fully resolved by physical characteristics of the human and environment without involving any cognitive processing. This is an important consideration that distinguishes perceived affordance in HCI from Gibsonian affordance in ecological psychology: from the cognitive, HCI perspective, an affordance exists conceptually to the human operator; from the ecological perspective, an affordance exists physically in the human operator's environment, regardless of what is believed.

In safety critical systems, specific physical manipulations may be necessary to configure the system, prevent hazardous situations, or recover from them, rendering Gibsonian affordance an important concept in this design space. For example, the fuselage door in a passenger aircraft cannot be opened when the aircraft exceeds a certain altitude if the pressure differential is too great for a human to pull the door inward. This is an important safety feature of the fuselage door, and the Gibsonian definition is sufficient for characterizing it.

In human-interactive systems having configurable hardware (e.g. cables with input/output connections), the designer must consider what affordances should emerge and when such that the system can be configured safely in the operational environment. In support of ensuring that the designed system is usable, analysts could benefit from a formal modeling and verification methodology. As discussed in Chapter 2, to support the development of formal models the analyst needs a formalism and a modeling technique; and to support formal verification analyses, the analyst could benefit from temporal logic specifications and a model checking technique.

Researchers in ecological psychology have developed a variety of formalisms for representing Gibsonian affordance [1, 2, 9, 3, 4, 6, 7]; however, there are limited approaches supporting the analyst in instantiating a formalism and verifying usability-related specifications involving affordance. One such approach is provided in this chapter. A review of extant affordance formalisms in Section 6.1

aids in identifying minimal requirements of a modeling technique and encoding tool for instantiating them. A tool and technique developed to meet these requirements are presented in Section 6.3. A verification methodology developed to enable model checking analyses of accuracy and error tolerance with respect to formal models is described in Section 6.3.14. The approach is demonstrated in a case study based on a medical device adverse event, and a scalability evaluation is conducted in Section 6.5. Discussions of the case study, the scalability evaluation, methodological considerations, and directions of future work follow.

6.1 Affordance Formalisms

Formalisms have been developed to represent Gibsonian affordance within HES using different combinations of variables, functions, and interpretations of the original theory. This subsection provides a verbal description of existing formalisms and the mathematical descriptions as-presented for each. Papers were selected from a Google Scholar search for the keywords “affordance,” “formalism,” and “Gibson.” The search returned 973 results. From these results, nine papers were selected using the following criteria:

- Published in a referred journal or conference proceedings
- Provides a symbolic, mathematical formalism for Gibsonian affordance
- Provides an original formalism (i.e., papers validating or applying an existing formalism were excluded)
- Is intended for application in human-environment systems (i.e., robotics papers were excluded)

6.1.1 Shaw and Turvey [1]

This formalism (called a coalitional model) extends Gibson’s definition by introducing the term “effectivity” to represent a human operator’s capabilities. Here, an effectivity is a dispositional property of a human, which means that under the right conditions, an effectivity is always actualized (i.e., an action is always taken). An affordance is a dispositional property of the environment, which means that under the right conditions, an affordance is always actualized, or an object in the

environment is always acted upon. Shaw and Turvey represent affordance and effectivity as one-to-one, complementary properties of the environment and human respectively. If an affordance and effectivity occur at the same time, the human operator must act.

The model provides a relational structure of the HES as a hierarchical set of sets (6.1) [1] expressed using four "grains" of analysis: basis (6.1a), relation (6.1b), order (6.1c), and value (6.1d). Here, X is a set of variables that describe the environment and Z is a set of variables that describe the human operator. Each grain is the Cartesian product of the previous grain, except V, which is the Cartesian product of O and a set $\{+,-\}$. The formalism provides no formal interpretation of V, but its value specifies which action a human must take.

$$HES = \begin{cases} B = (X, Z) & \text{(6.1a)} \\ R = B \times B & \text{(6.1b)} \\ O = R \times R & \text{(6.1c)} \\ V = O \times \{+,-\} & \text{(6.1d)} \end{cases}$$

To inform the development of a new formalism (discussed next), A.J. Wells has explained each element of the coalitional model intuitively [2]:

B: The basis grain describes the set of variables over which the model is defined

R: The relation grain describes the ecological relations that are possible given the basis variables

O: The order grain provides descriptors for the affordance structure of the environment and for the effectivity structure of an animal

V: The value grain specifies which affordances are noticed or which effectivities are activated on a given occasion

6.1.2 Wells [2]

Wells integrates Gibson's theory of affordance with Turing's theory of computation to develop a formalism that captures the temporal ordering of human actions within a Turing machine represen-

tation [183] (6.2). Here, symbolic variables represent a single human operator, actions that can be executed, and affordances/effectivities using two respective tuples. The first tuple (6.2a) defines an affordance (A) at the intersection of two variables that represent properties of the human operator (q) and the environment (a). The second tuple (6.2b) defines an effectivity at the intersection of three variables that represent a human behavior (b), the next-state of the human operator (p), and a description of the human behavior (k). The temporal ordering of human operator states is defined over the set of transitions between each instance of b and p.

$$A = (q, a) \tag{6.2a}$$

$$E = (b, p, k) \tag{6.2b}$$

6.1.3 Turvey [3]

In (6.3), aspects of the coalitional model [1] are leveraged within a new, input/output function-based formalism. Here, affordances are dispositional properties of the environment and effectivities are dispositional properties of the human operator. A dispositional effectivity is also describable as a human action that always executes successfully, where "successfully" does not refer to what the human desires (which depends partly on cognition), but rather the post-action conditions that manifest immediately the human actualizes a juxtaposed affordance and effectivity [3]. Mathematically, X represents the environment with p affordances; Z represents the human operator with q effectivities; and W represents the human-environment system with pq properties. The property r is the co-occurrence of p and q in W_{pq}. The juxtaposition function j provides values of r. The juxtaposition function may also provide zero or more possible human actions that can execute within W_{pq}, where if multiple actions are available, exactly one must execute.

p is an affordance of X and q an effectivity of Z iff:

$$W_{pq} = j(X_p, Z_q) \text{ possesses } r \tag{6.3a}$$

$$W_{pq} = j(X_p, Z_q) \text{ possesses neither } p \text{ nor } q \tag{6.3b}$$

$$\text{Neither } Z \text{ nor } X \text{ possesses } r \tag{6.3c}$$

6.1.4 Stoffregen [4]

Stoffregen modifies Turvey's formalism in three ways: he redefines affordance as an emergent property of the animal-environment system h, rather than as a dispositional property of the environment, which precludes the need for an action to execute when it becomes available; he removes the juxtaposition function and replaces it with a tuple (6.4a), which prevents the formalism from producing a case in which multiple affordances are actualized simultaneously; he formalizes human behavior with a psychological choice function that is separate from the formalism, which selects a single action based on the human operator's intention (not discussed in this work). In (6.4), the affordance h is a property of the HES defined by the relation between p and q [4].

h is an affordance of W_{pq} iff:

$$W_{pq} = (X_p, Z_q) \text{ possesses } h \tag{6.4a}$$

$$\text{Neither } Z \text{ nor } X \text{ possesses } h \tag{6.4b}$$

6.1.5 Thiruvengada and Rothrock [5]

In this formalism, researchers combine the mathematical notations used by Turvey [3] and Wells [2]. The accompanying modeling technique in [184] uses Colored Petri nets [185] to model multiple human operators, concurrent affordances, possible human actions, discrete time and space, and goal-oriented action selection. The Discrete Event System Specification (DEVS) [186] formalism is

applied to simulate changes to model variables within the HES. In (6.5), possible human actions (PA) exist when an affordance (P) and an effectivity (Q) also exist (6.5a), and they are juxtaposed in the HES (W_{pq}) (6.5b) at the same time and spatial location (6.5c). Properties of the HES are provided by sensory functions that represent three respective ways in which humans sense their environment. The three types of properties are specified as φ_i in φ, where i can be v for visual properties, a for auditory properties or h for haptic properties. In (6.5d), φ_i provides a set of possible actions (6.5d). Thus, theories of affordance from HCI and ecological psychology are integrated in a way that incorporates perceptual, motor, and cognitive functions. In (6.5e), φ_i provides a set of effectivities. In (6.5f), φ provides a set of affordances. A single action in PA is then executed in the model based on an analyst-defined sequence of actions that proceeds toward a goal for the human operator.

$$\exists\, PA \leftrightarrow \exists\, P \text{ and } \exists\, Q \text{ iff:} \tag{6.5a}$$

$$j(X_p, Z_q) = W_{pq}, \text{ and} \tag{6.5b}$$

$$X \text{ and } Z \text{ exist at the same time and space, where} \tag{6.5c}$$

$$f_\varphi : \cup_i\{\varphi_i : W_{pq}\} \rightarrow PA, \tag{6.5d}$$

$$f_\varphi : \cup_i\{\varphi_i : X_p\} \rightarrow P, \text{ and} \tag{6.5e}$$

$$f_\varphi : \cup_i\{\varphi_i : Z_q\} \rightarrow Q, \tag{6.5f}$$

6.1.6 Greeno [6]

Instead of leveraging discrete mathematics (as in the formalisms discussed thus far), Greeno's formalism draws on situation theory [187], where a situation is the collection of visual objects, their properties, and their relations to one another in a particular time and place [187]. The term *ability* is used in place of *effectivity* to define motor capability features of a human operator. To formalize affordance, he uses a logical proposition (6.6) and a verbal description (in lieu of a mathematical description) to state the HES conditions enabling it. Here, affordances are situation-dependent preconditions for a human operator to execute some action in a way that achieves "good effects" (6.6)

[6]. Semantically, what makes an effect "good" is not explained in [6]; and while the temporal relationship between "action by agent" and "good effects" is assumed to described a temporally ordered relationship between "action" and "effects," such a relationship is not defined mathematically by the logical implication operator in (6.6).

$$\langle\langle \text{action by agent} \rangle\rangle \Rightarrow \langle\langle \text{good effects in situation} \rangle\rangle \tag{6.6}$$

6.1.7 Chemero [7]

Combining the syntax of discrete math with the semantics of situation theory, an affordance in Chemero's formalism is the feature of a situation that emerges from the relation between a human operator and the environment, where the environment includes the physical object(s) to be acted upon. This formalism draws on the same definition of "situation" as Greeno [187, 6] and specifies affordance as a logical expression (6.7). Chemero states that a *feature* describes a situation in the environment and *ability* is a capability of the human operator. An ability is distinct from Turvey's [3] dispositional effectivity because a human ability can fail while a dispositional effectivity cannot [7].

$$\text{Affords-}\phi\,(feature, ability) \tag{6.7}$$

6.1.8 Lenarčič and Winter [8]

In [8], researchers integrate situation theory [187] and object-oriented programming within a hierarchical formalism representing affordances, environments, and human operators (6.8). Each affordance (ϕ) is an object of the form shown in (6.8a), where s is a situation, ψ is a human action, and i is an individual (e.g. a human operator). The expression in (6.8b) specifies that an action must come from a set of possible actions **ACT** (not defined mathematically in [8]), and the expression in (6.8c) specifies that the situation must be provable in the environment (e), which comes from a set of environments **Env**. A human operator is composed of a name (x), an ability (a), and a niche

(n). A niche is defined as a set of abilities that the human operator has based on her skills and experiences. For example, if the human operator is a manned aircraft pilot, one ability in her niche could be "land plane." The researchers define semantics for specifying situations that can be proven to exist in a environment based on physical characteristics of human operators and objects therein. Further details can be found in [8].

$$\phi = \langle\langle \Phi, s, i \rangle\rangle, \text{ where} \tag{6.8a}$$

$$\Phi \in \mathbf{ACT}, \tag{6.8b}$$

$$s \vdash e \in \mathbf{Env}, \text{ and} \tag{6.8c}$$

$$i = \langle\langle x, a, n \rangle\rangle \tag{6.8d}$$

6.1.9 Warren [9]

Warren's formalism diverges from those discussed thus far by leveraging numerically valued variables instead of discrete mathematical symbols to define affordance. This enables the analyst to measure affordances and effectivities using empirical studies. A ratio of variables describing properties of the environment (E) and properties of the human operator (A), where an effectivity (6.9b) is the inverse of an affordance (6.9a). In (6.9), π is a system of equations in which each element π_i in π is used to define a different ratio of a property of the environment and a property of the human operator [9]. For example, a π system that describes a human operator's ability to grasp a door handle may include an equation $\pi_1 = W_d/W_h$, where W_d is the width of the door handle and W_h is the width of the human operator's hand in centimeters. This formalism is unique in that it is the only one of the nine discussed in this section that specify measurements of the human and environment directly.

$$\pi_{\text{affordance}} = E/A \tag{6.9a}$$

$$\pi_{\text{effectivity}} = A/E \tag{6.9b}$$

6.2 Representing Affordances Formally

In the spirit of formal methods, each of the discussed affordance formalisms could be instantiated to support formal modeling and verification of human-interactive systems (specifically, configurable hardware components and end-user opportunities to manipulate them). However, extant formalisms employ discrepant theories, terminologies, and mathematical representations. Additionally, it is unclear if any of them can be instantiated and verified in a way that is both theoretically and mathematically correct. Despite these challenges, analysts could benefit from a modeling technique that is sufficient for instantiating one or more extant formalisms. An encoding tool could also reduce the need for combined expertise in ecological psychology and formal methods. The requirements listed in this section begin to capture what is needed of such a technique and accompanying tool.

6.2.1 Requirements of a Modeling Technique

Considering the HES properties that shape affordance, five elements must be considered in a modeling technique for instantiating an extant formalism: physical objects in the operational environment, human operators in the operational environment, spatial relations among objects and human operators, motor capabilities of human operators to physically manipulate objects, and temporal evolution of the HES. Minimal requirements for such a technique are listed below.

1. The modeling technique should enable the analyst to represent physical objects

All of the discussed formalisms include a means of representing an HES, which may include one or more physical objects. The technique should therefore support the analyst in representing at least one physical object. In software and systems engineering, analysts use hierarchical-heterarchical modeling languages to represent physical objects [188]. These techniques allow analysts to specify types of system elements and the part-whole relationships among them. For example, analysts can use a hierarchical-heterarchical structure to specify types of vehicles as cars and motorcycles. They can then specify sub-components that exclusively belong to cars, such as doors, and sub-components that exclusively belong to doors, such as knobs. A similar method can be applied for specifying physical entities within HES to help analysts identify and model the specific parts involved

in affordances.

2. The modeling technique should enable the analyst to represent human operators

All of the discussed formalisms have symbols and semantics for representing human operators as part of an HES. Therefore, the modeling technique should support the analyst in representing a human operator.

3. The modeling technique should enable the analyst to represent environmental conditions that effect emergent affordances

The discussed formalisms provide various symbols and semantics for representing conditions in the environment that effect emergent affordances, such as the spatial relationship between two objects. It is therefore necessary for a modeling technique to provide a way of representing such conditions.

4. The modeling technique should provide a way of representing temporal evolution of conditions in the HES

All of the discussed formalisms are based on Gibson's definition, which defines affordances as emergent in a temporally evolving HES. Human motor capabilities are considered static, while spatial relations evolve. The modeling technique should therefore provide a way for the analyst to represent how spatial relations in the HES evolve over time.

5. The modeling technique should enable the analyst to represent motor-action capabilities of the human operator

All of the discussed formalisms include verbal and/or mathematical representation of human motor-action capabilities. These semantics employ Gibson's definition to characterize one affordance and one complementary effectivity or many abilities that could be "actualized" via one motor action. Thus, to support the analyst in instantiating an extant formalism, the modeling technique should provide a way of representing human motor-action capabilities that are needed to actualize an affordance by executing one or more motor actions.

6.2.2 Requirements of an Encoding Tool

To facilitate the process of verifying affordance-related characteristics of the interface, it could be beneficial to develop an encoding tool that supports the analyst. Such a tool should include a formal description language and an accompanying translator for generating model checking syntax. A translator should be capable of parsing instantiated formal descriptions and generating model checking syntax. A formal description language should facilitate the application of a modeling technique. Initial requirements for such a language are listed below.

1. The language should enforce good encoding practices that are utilized in engineering domains

One commonality among extant formalisms is that they are grounded in psychological theories, not engineering practices. Because one purpose of this work is to enable the practical application of affordance in HFE, the language should enforce a hierarchical-heterarchical structure that is common in engineering [189].

2. The language should provide a constrained set of keywords utilized in an extant affordance formalism

All of the discussed formalisms define affordance and human capabilities using common and discrepant keywords, terms and ideas. For example, in [3] the term “effectivity” is used to describe human capabilities that never fail, while in [4] the term “ability” is used to describe human capabilities that sometimes fail. Either term is used exclusively within mathematically similar yet theoretically conflicting formalisms. To support the analyst in applying a modeling technique to instantiate an extant formalism, the language should incorporate a set of terms that are consistent (i.e. either “ability” or “effectivity” should be utilized in the same instantiated description).

3. The language should provide keywords for representing spatial relations among human operators and objects in the environment

As discussed in Section 6.2.1, the modeling technique should support the analyst in representing spatial relations in three dimensions. To facilitate the process of applying such a technique, the language should provide corresponding keywords that can be unambiguously defined.

In natural languages, the spatial position of an object may be described in reference to another object [190]; for example, "The driver is inside the car." In Geographic Information Systems (GIS), formal techniques developed to verbally describe spatial relationships among objects in two dimensions follow the same convention [191, 192, 193, 194, 195]. In [195] and [196], two types of spatial relationships are identified: topological and directional. These descriptions can be leveraged within the language.

A topological relationship describes the connectedness of one object in reference to another [197]. This approach is based on a mathematical formalism that uses four criteria of spatial relationships for two-dimensional objects: (1) an intersection at the boundaries of two shapes, (2) common interior parts of two shapes, (3) the boundary of one shape as part of the another shape's interior, and (4) the interior of one shape as part of another shape's boundary [198]. Using these criteria, there are eight exclusive (i.e. only one can exist at any time) topological relations between two entities in two-dimensional space: disjoint, touch, equal, inside, contains, covered by, covers, and overlap. Further details on the mathematical and semantical differences between these relations can be found in [12].

To support the analyst in modeling topological relations, it could be beneficial to abstract possible keywords from these eight relations. Utilizing a natural language description, "covers" and "covered by" can be reduced to one relation; i.e., "A covers B" and "B is covered by A" have the same meaning. "Contains" and "inside" can be reduced to one relation in the same way; i.e., "A is inside B" and "B contains A" have the same meaning. "Contains" and "covers" can be reduced to one or the other, as one object that covers another also contains it. "Equal" is characterized by one object covering another object of the same size. Because the majority of extant affordance formalisms do not have semantics for specifying the size of objects (with [9] as the exception), "equal" can be abstracted as a kind of "covers." This leaves four relations, all of which can be utilized within the language (Fig. 6.1a–e).

A directional relationship describes the directional placement of one object in reference to another [196]. In [196], two-dimensional directional relationships are described using cardinal compass

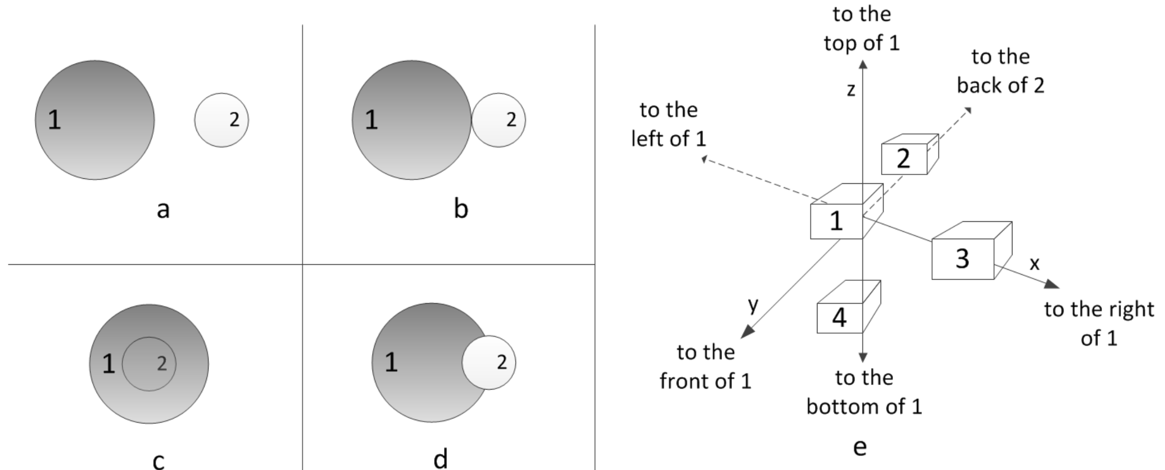


Figure 6.1: (a-d) Conceptual representations of two-dimensional topological relationships inspired by the formal language described in [12]: (a) 1 is disjoint to 2, (b) 1 touches 2, (c) 1 covers 2. (d) 1 overlaps 2. (e) A graphical representation of the topological-spatial relations: 2 is to the back of 1, 3 is to the right of 1, and 4 is to the bottom of 1, inspired by the techniques described in [13] and [14].

directions north, south, east and west. In [199], two-dimensional directional relationships can be described from the perspective of a human observer using terms such as *left-of* and *right-of*. Methods developed in civil engineering extend GIS techniques by incorporating directional keywords that support describing objects in three dimensions [200]. In [200], compass directions are used to describe directional relations, where north and south are defined on the x-axis and east and west are defined on y axis. Additional keywords for describing an object positioned above or below a reference object are defined on the z-axis. In virtual reality, a different formal language has been developed to support semantic modeling of virtual environments in three dimensions [13]. This language allows spatial relations to be described relative to a reference object from the perspective of a human operator rather than as compass directions. It employs phrases such as *to the front of* and *to the left of*. This directional relation terminology (Fig. 6.1e) could be useful within an affordance modeling technique.

4. The language should provide keywords for representing human operator motor capabilities

As discussed in Section 6.2.1, the modeling technique should support the analyst in representing human motor capabilities. In engineering domains standard terminology is utilized for describing

movement of objects in three dimensions. One such standard is the six degrees of freedom of a rigid body in three-dimensional space (6DoF) [201]. There are two general types of motions that can be described using 6DoF, positional changes and rotational changes. Positional changes reflect a change in the placement of an object in 3-D space, typically using the object's center (or origin) as a reference point. Rotational changes reflect a change in the orientation of an object around its center (or origin). Specific motions are defined along three axes (x, y and z). Leveraging this standard, the language could provide keywords for specifying motions along each axis. For positional changes along the x-axis, the terms *translate left* and *translate right* may be used to describe leftward and rightward movement respectively. For positional changes along the y axis, the terms *position back* and *position forth* may be used. For positional changes along the z axis, the terms *position up* and *position down* may be used. For rotational changes along the x-axis, the terms *yaw right* and *yaw left* may be used. For rotational changes along the y-axis, the terms *roll right* and *roll left* may be used. For rotational changes along the z-axis, the terms *pitch back* and *pitch forth* may be used.

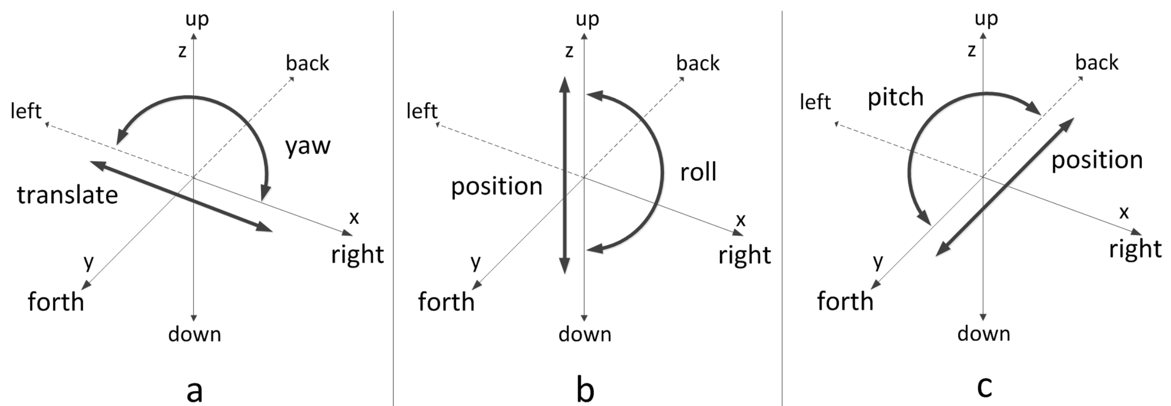


Figure 6.2: A visual representation of positional and rotational movements based on the six degrees of freedom of a rigid body in three dimensional space. Rotational movements are indicated by the arcing arrows. Positional movements are indicated by the straight arrows.

6. The language should support parsing capabilities

As discussed earlier in this section, a translator could facilitate the development of formal models that are amenable to formal verification. It is therefore necessary that the language enables parsing capabilities, similar to the languages developed by other researchers for task analytic applications (e.g. EOFM-XML [10]).

6.3 The CAVEMEN Approach

The approach discussed in this section intends to support requirements listed in Section 6.2. A modeling methodology includes an XML-based [71] language for instantiating one of three extant affordance formalisms. An automated translation tool parses instantiated XML representations and generates formal affordance models in the syntax of SAL [68]. Utilizing an instantiated SAL model, an accompanying modeling technique supports the analyst in specifying initial end-user motor capabilities and evolving spatial relations among entities in the environment. Together, these elements constitute the **C**apability, **A**ffordance, and e**V**olving r**E**lations **M**od**E**li**N**g (CAVEMEN) approach. It includes:

1. An XML-based grammar and affordance modeling technique for specifying objects and affordances in a human-environment system
2. An automated translation tool that converts an CAVEMEN-XML representation to a formal model in the syntax of SAL
3. An HES modeling technique for specifying end-user motor capabilities and evolving HES configurations
4. A verification methodology, including LTL specifications and a model checking technique for verifying error-tolerance and accuracy of configurable hardware

Leveraging the general theory of Gibsonian affordance [82], the CAVEMEN approach enables the analyst to specify an affordance that emerges when any HES configuration satisfies a set of properties concurrently. As in many of the discussed formalisms, when an affordance emerges it can be actualized in one human action, such as turning a steering wheel [202] or catching a ball [9]. Concurrency reflects the characteristic of affordance as "pointing both ways," where characteristics of the human operator (i.e. motor capabilities) and characteristics of the environment (i.e. spatial relations) are inseparable with respect to temporally emergent properties of affordance. These properties are represented in two ways:

1. Evolving spatial relations among environmental entities, including objects and human operators, which are defined based on the initial and future HES configurations that are possible

2. Static motor capabilities required of a human operator to move objects, which are defined by how a human operator may need to manipulate an object for an affordance to emerge in any possible HES configuration

The formal semantics of CAVEMEN-XML, which are are specified using the XSD standard [159], are described in the remainder of this section. The modeling technique for instantiating a CAVEMEN-XML model is also defined.

Figure 6.3: Visual representation of the CAVEMEN-XML grammar. In a–n, rectangles are nodes, rounded-edge rectangles are attributes, and arrows point to child nodes. Sibling nodes are horizontally adjacent. Required nodes and attributes are outlined in dotted red lines. For sibling nodes, a blue dashed outline indicates that at least one of either is required. Groupings of nodes within shapes correspond to the subsections describing them in Section 6.3.

6.3.1 The Root Node

The root node of a CAVEMEN-XML specification is named *hes* (Fig. 6.3a), which stands for human-environment system. It contains variables that represent physical objects and affordances. Variables representing these elements are specified within *modelobject*, *subobject*, *atomicobject*, and *affordance* child nodes.

6.3.2 Model Objects, Subobjects, and Atomic Objects

Each *modelobject* node (Fig. 6.3b) represents a physical object in the system that has one or more parts making up its whole composition. Each *modelobject* may be decomposed into zero or more *subobject* (Fig. 6.3c) child nodes, which represent its parts. A part can be anything physically inseparable from the whole component such as a specific interior/exterior surface and a permanently attached segment or widget. For each *subobject* node, the analyst can choose to decompose it into another *subobject* or into an *atomicobject* node (Fig. 6.3d), which cannot be decomposed further. An *atomicobject* node can be declared on its own to represent an object with no part-whole composition. All three of these nodes must be assigned a unique, descriptive *name* attribute, which should help to identify it.

The door example mentioned at the outset of this chapter can be specified in CAVEMEN-XML as shown below. One *modelobject* node is encoded to specify the door (*mDoor*); one *subobject* node specifies that the knob (*sKnob*) is a permanently attached part of the door; and one *atomicobject* node specifies that the lock (*aoLock*) is a permanently attached part of the knob.

```
<modelobject name="mDoor">
 <subobject name="sKnob">
  <atomicobject name="aoLock"/>
 </subobject>
</modelobject>
```

6.3.3 Affordance

Each *affordance* node (Fig. 6.3e) represents an affordance to be specified according to one of three extant formalisms. The node must be assigned a unique, descriptive *name* attribute, which should identify the affordance being specified, and a *formalism* attribute, which identifies the terminology

and theory that the specification will employ. The *formalism* attribute can be valued *stoffregen*, *greeno* or *chemero*. If the analyst specifies multiple *affordance* nodes, each node should have the same *formalism* attribute value to ensure that the same underlying theory and formalism are utilized consistently throughout the model. Different techniques for instantiating each formalism are described in Section 6.3.7. For any instantiated formalism, each affordance node can have one or more *humanoperator* nodes.

6.3.4 Human Operator

Each *humanoperator* node (Fig. 6.3) represents a human having capabilities to perceive and manipulate objects in a HES. Each human operator must be assigned a *name* attribute. The *name* attribute is necessary for specifying spatial relations among human operators and physical objects using *relation* nodes, which will be described next. Each *humanoperator* node contains one or more *relation*, *component*, *subcomponent* and *atomcomponent* child nodes.

6.3.5 Relation

Each *relation* node (Fig. 6.3g) is used to specify one spatial relationship between a model entity with respect to an associated entity from the human operator's initial perspective (what is meant by "initial perspective" is discussed later in this section). Each *relation* node must be assigned a *topology* attribute having one value from Table 6.1. The attribute value identifies the topological relationship among two objects that is needed for an affordance to emerge. An optional *direction* attribute represents one surface of an *associate* (discussed next), and it can be assigned a value from Table 6.2. Omitting a *direction* attribute is equivalent to specifying all six directions.

An *associate* attribute represents a model entity with respect to which other attributes of a *relation* node are defined; e.g., the *component* is covering the top of its *associate*, where the top is a two-dimensional surface. Once defined, the identities of *associate* entity surfaces do not change; i.e., the front-facing surface is always referenced as such (see for example Fig. 6.6a), even if the human operator's perspective could change. This is what is meant by the human operator's "initial perspective."

These semantics help ensure that evolving spatial relations can be modeled unambiguously. For example, the front of a car is always the front, no matter how the human operator views it. The left and right sides (surfaces in CAVEMEN-XML) could then be defined invariantly with respect to the car's front-facing surface, even if the human operator changes her position with respect to the car such that the left- and right-side surfaces are reserved. If the entity has identical surfaces along all axes of symmetry, such as a sphere or cube with no exterior markings, the analyst could use *direction* attributes interchangeably, since no directions are distinct and there would be no semantic difference to a human operator.

Each *relation* node may be assigned an optional *condition="not"* attribute, which specifies that the relation must not exist. All child *subcomponent* and *atomcomponent* nodes inherit *relation* nodes from higher level *component* and *subcomponent* nodes, except if they are referenced within *associate* attribute values. For example, consider a *component* node having one *relation* child node and one *subcomponent* child node. If the *relation* node references the *subcomponent* node's name within its *associate* attribute, the *subcomponent* does not inherit it, since such inheritance would constitute a self-relation (i.e. the *subcomponent*'s associate is itself). If the analyst encodes self-relations they are ignored by the XML-to-SAL translation tool (discussed later).

Table 6.1: Topology attribute values. All topological relations are mutually exclusive

Value	Semantics
disjoint_to	There is empty space between one or more surfaces of an entity and one or more surfaces its associate
touching	One or more surfaces of an entity touch one or more surfaces of its associate (without overlapping or covering)
covering	One or more surfaces of an entity completely cover one or more surfaces of its associate. If it is not completely covering, but there is contact, then the relation must be either touching or overlapping.
overlapping	One or more surfaces of an entity overlap one or more surfaces of its associate. If there is contact, but the surfaces are not overlapping, then the relation must be either touching or covering.

Table 6.2: Direction attribute values

Value	Semantics
right_of	Surface right of associate's origin
left_of	Surface left of associate's origin
front_of	Surface front of associate's origin
back_of	Surface behind associate's origin
top_of	Surface above associate's origin
bottom_of	Surface below associate's origin

6.3.6 Component, Subcomponent, and Atom Component

Component (Fig. 6.3a), *subcomponent* (Fig. 6.3b) and *atomcomponent* nodes (Fig. 6.3c) are used to specify spatial relationships among physical objects with respect to each other from the human operator's perspective (discussed further in Section 6.3.5). Each must be assigned a *name* attribute that references a *modelobject*, *subobject*, and *atomicobject* respectively. Each *component*, *subcomponent* and *atomcomponent* node must also contain one or more *relation* nodes and/or exactly one *ability* node (discussed below).

6.3.7 Ability

The *ability* node is utilized to specify motor capabilities of the human operator according to the formalisms developed by Stoffregen [4], Chemero [7], and Greeno [6]. *Ability* is chosen over *effectivity* because the underlying theories employing *ability* do not require one-to-one complementarity of affordances and motor capabilities. This is advantageous for affordance specification because real HES affordances could involve multiple, concurrent abilities; as in the "door openability" example, where a human operator must be able to turn the knob (one ability) and pull the door open (a second ability).

Each *ability* node (Fig. 6.3k) represents a human capability to move a *component*, *subcomponent* or *atomcomponent* in three-dimensional space of the operational environment. As in all three affordance formalisms that leverage this keyword, each *ability* node specifies a motor capability that is required of the human operator, independently of other objects, agents, and conditions in the HES

(i.e. independently of all *relation* nodes and other *ability* nodes). This convention is based on an overarching assumption of Gibsonian affordance: human motor capabilities are a characteristic of the human operator alone, and while they are inseparable from affordance, they persist independently of the environment [82].

As discussed in Section 6.1, one way researchers have characterized human capabilities formally with respect to affordance is by defining static parameters such as leg length and palm width [9]. In CAVEMEN-XML, these characteristics are defined indirectly with respect to a single object that the analyst has defined within a *component*, *subcomponent* and *atomcomponent* (referencing an *object*, *subobject*, or *atomicobject* node respectively). For example, consider an *affordance* node named "*DoorOpenable*, a direct child *component* node named "*Door*," and a grandchild (i.e. a child of *Door*) named "*Knob*." One way to specify an *ability* child node of "*Knob*" involves reasoning about how large the human operator's hand needs to be and how much force is required to rotate the knob in either direction, where either direction could be necessary for the door to be openable. Regardless of where the door is in relation to the human operator, whether it is locked, and how many other objects/agents could be in the way, the motor capability of rotating the knob must persist in any situation for the affordance to emerge. Thus, the *ability* node is utilized to identify these needs, and it should be assigned a *name* attribute accordingly, such as "*TurnKnob*."

The exact movements in three-dimensional space defining a motor capability can be specified using the child nodes *translatable*, *positionable* and *orientable* (discussed next).

6.3.8 Translatable, Positionable and Orientable

These nodes (Fig. 6.3l–n) represent the six degrees of freedom of a rigid body in three-dimensional space (Fig. 6.2 and 6.3). Every attribute in each of these nodes (if specified) must be assigned a Boolean value, which represents whether or not the human operator might need to perform the movement. Omitting one or more nodes and/or attributes is equivalent to specifying that *true* and *false* are both applicable. This syntax is utilized to specify the underlying composition defining a necessary human motor capability with respect to:

- The origin (i.e. the center) of the object that must be moved, independently of its parent

object (explained further below)

- The motor capability being defined (i.e. the *name* attribute of the direct parent *ability* node)

For example, consider the affordance of "*DoorOpenable*" and the ability of "*TurnKnob*" mentioned in Section 6.3.7. Again, without considering where the human operator and door are in relation to each other, whether the door is locked, or whether it is functioning correctly, the ability of rotating the knob clockwise *or* counterclockwise could be needed to ensure that the door is openable in any potential scenario. An alternative way of reasoning about this involves considering all imaginable HES configurations, including all possible spatial relationships among the knob and human operator. While there are many situations in which the door is openable without turning the knob, such as if its bolt is missing, there are also many in which the knob must be turned. In accordance with Gibson's definition, all of them must be considered.

Utilizing the formal semantics of CAVEMEN-XML (based on the 6DoF standard), movements for the motor capability of "*TurnKnob*" can be specified within an *orientable* node (Fig. 6.3l) as shown below.

```
<ability name="TurnKnob">
 <orientable roll-right="true" roll-left="true"/>
</ability>
```

The valued attributes *roll-right="true"* and *roll-left="true"* correspond to the clockwise and counterclockwise movements mentioned earlier. Other nodes and attributes are not specified, as it is irrelevant if the human operator can position, translate, or orient the knob in any other way.

6.3.9 Translation to SAL

A custom, JavaScript-based translator supports model checking analyses of CAVEMEN-XML representations. It parses nodes of an instantiated CAVEMEN-XML representation to generate a formal model in the syntax of SAL [68], incorporating all entities and affordances in the HES. A translated formal model is a SAL context (Chapter 3, Section 3.2) named `cavemen` by default, and it has enumerated types, record types, and array types representing all spatial relations and human operator motor capabilities specified in the instantiated CAVEMEN-XML representation. Affordances are

represented within a module named `affordance` and the HES is represented within a module named `HES`. A general representation of an automatically generated SAL context is shown below. Model infrastructure to be discussed throughout the section is marked with text in parentheses referencing the subsection in which it is discussed. The asynchronous composition of affordance and HES modules ensures that changes to spatial relations among HES entities update in the next-state when an affordance is actualized (explained in Section 6.3.13).

```
cavemen:  CONTEXT =
BEGIN
(Automatically generated types, Section 6.3.9.1)
   affordance:  MODULE =
   BEGIN
   (Affordance module variables, Section 6.3.9.2)
     DEFINITION
     (Affordance module definitions, Section 6.3.11)
   END;
   HES: MODULE =
   BEGIN
   (HES module variables, Section 6.3.10)
   END;
   affordances:  MODULE = affordance [] HES;
END
```

6.3.9.1 Automatically Generated Types

Spatial relation and motor capability values are enumerated types generated from *topology* and *direction* attributes as well as *ability* child node attributes. Spatial relation enumerated types could include all values from Tables 6.1 and 6.2 (as shown below). Motor capability enumerated types are generated for all movements represented in Fig. 6.2.

```
topological:  TYPE = {disjoint_to, touching, covering, overlapping};
directional:  TYPE = {right_of, left_of, front_of, back_of, top_of, bottom_of};

orient     :  TYPE = {pitch_back, pitch_forth, yaw_right, yaw_left, roll_right,
                      roll_left};
position :  TYPE = {up, down, back, forth};
translate:  TYPE = {left, right};
```

Spatial relation enumerated types `topological` and `directional` are incorporated within an array type of six Boolean-valued directions each having a topological relation. Motor capability

enumerated types `orient`, `position`, and `translate` are incorporated within a record type having three arrays of Boolean-valued movements.

```
relations:  TYPE = ARRAY directional OF topological;
abilities:  TYPE = [#orientable  :  array orient    of boolean,
                     positionable:  array position  of boolean,
                     translatable:  array translate of boolean#];
```

An additional set of record types is generated to represent perceivable spatial relations among environmental entities, including objects and human operators (represented within CAVEMEN-XML *relation* nodes). The translator generates one record type for each entity (*humanoperator*, *component*, *subcomponent*, or *atomcomponent*) having one or more *relation* child nodes. The name of each record type is the entity's *name* attribute with `_rels` added at the end. Each *associate* attribute (i.e. an associated entity) is represented as an identifier having the record type `relations`. A general encoding is shown below for a CAVEMEN-XML representation having n entities, all of which are considered *atomcomponent* nodes. Each has m *relation* child nodes, all of which have *associate* attributes referencing other *atomcomponent* nodes.

```
aoEntity_1_rels:  TYPE = [#aoAssociate_1:  relations,..., aoAssociate_m:
relations#];
      ...
aoEntity_n_rels:  TYPE = [#aoAssociate_1:  relations,..., aoAssociate_m:
relations#];
```

6.3.9.2 Affordance Module Variables

The translator generates a representation of all HES affordances with a module named `affordances`. The module has input variables representing spatial relations among HES entities, each with a corresponding record type. Each input variable is the *name* attribute of an HES entity. A general encoding of input variables is shown below, where each variable's type is leveraged from general record types `aoEntity_1_rels`, ..., `aoEntity_n_rels` described in the previous section (Section 6.3.9.1).

```
INPUT aoEntity_1:  aoEntity_1_rels
        ...
INPUT aoEntity_n:  aoEntity_n_rels
```

Input variables are also generated to represent human operator capabilities, each having the record type `abilities`. The translator parses *humanoperator*, *component*, *subcomponent*, *atom-*

component, and *ability* nodes and generates one input variable for each human capability to move another entity. Each record type is named using the *humanoperator* node's *name* attribute, followed by an underscore, followed by the *name* attribute of a *component*, *subcomponent*, or *atomcomponent* node. The general SAL syntax shown below represents one *humanoperator* node named "pHuman" having *n atomcomponent* child nodes, each with one *ability* child node.

```
INPUT pHuman_aoEntity_1:  abilities
        ...
INPUT pHuman_aoEntity_n:  abilities
```

6.3.10 HES Module Variables

The `affordance` module inputs specifying spatial relations and human capabilities are outputs of an automatically generated `HES` module representing the human-environment system. The contents of this automatically generated module are always the input and output variables encoded generally below representing affordances and HES entities respectively. The type of each affordance input variable depends on the instantiated formalism (discussed in Section 6.3.12).

```
INPUT affordance_1 ... affordance_n:  ...
OUTPUT entity_1:  entity_1_rels
        ...
OUTPUT entity_n:  entity_n_rels

OUTPUT pHuman_aoEntity_1:  abilities
        ...
OUTPUT pHuman_aoEntity_n:  abilities
```

Utilizing output variables, the analyst can manually add infrastructure specifying an initial HES configuration as well as next-state configurations that evolve over time. Such infrastructure should represent initial human capabilities, initial spatial relations, and transitions to spatial relations that evolve based on what affordances exist, where affordance input variables are leveraged within guarded transitions. Techniques for encoding these initializations and transitions are discussed in Section 6.3.12.

6.3.11 Affordance Module Definitions

The module's remaining syntax depends on the extant affordance formalism and underlying theory represented in CAVEMEN-XML (specified in the *affordance* node *formalism* attribute). An example

affordance node (Fig. 6.4a) illustrates the automatically generated model infrastructure corresponding to each possible *formalism* attribute value: *stoffregen* (Fig. 6.4b), *greeno* (Fig. 6.4c), or *chemero* (Fig. 6.4d).

The CAVEMEN-XML *affordance* node represented in Fig. 6.4a specifies an HES including a human operator (*pHuman*), a book (*aoBook*), and a table (*aoTable*). One affordance is specified as the human operator's opportunity to place the book on the table's top surface (*BookPlaceableOnTable*). In this example, the book is placed on the table if the top-side surface of the table is completely covering one or more surfaces of the book (i.e., the book cannot be hanging off the edge of the table). The *relation* node specifies that the book is not already placed on the table, and the *ability* node specifies that the human operator must be capable of moving the book left, right, up, down, back, and forth as well as rotating the book left or right.

When the *formalism* attribute is "stoffregen" (Fig. 6.4b), the translator generates model infrastructure corresponding to Stoffregen's formalism (6.4) [4]. This includes one Boolean local type variable corresponding to properties of the environment:

- X_p in Stoffregen's formalism (6.4)
- `X_`, followed by the *affordance* node's *name* attribute in SAL (Fig. 6.4b, blue, dashed-line rectangle)

Its value is *true* if spatial relations in the HES satisfy those represented within CAVEMEN-XML *affordance* nodes (Fig. 6.4b, blue text). The translator also generates one Boolean type local variable corresponding to properties of the human operator:

- Z_q in Stoffregen's formalism (6.4)
- `Z_`, followed by the *ability* node's *name* attribute in SAL (Fig. 6.4b, green rectangle)

Its value is *true* if human capabilities in the HES satisfy those represented within *translatable*, *positionable*, and *orientable* nodes (Fig. 6.4b, green text). The output is a pair of Boolean type variables corresponding to the affordance:

- $W_{pq} = (X_p, Z_q)\, possesses\, h$ in Stoffregen's formalism (6.4)

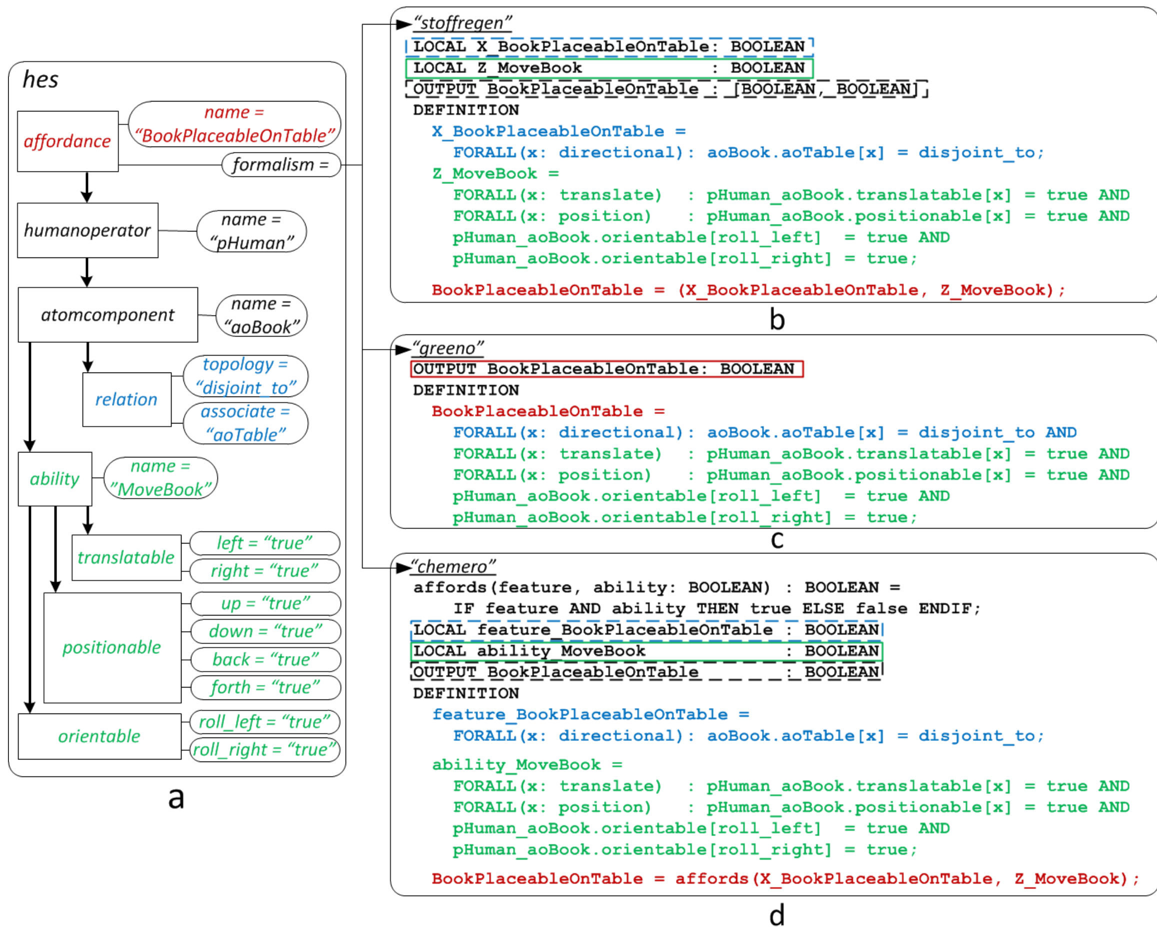


Figure 6.4: A graphical representation of an example CAVEMEN-XML *affordance* node and translated SAL model infrastructure for three extant affordance formalisms. (a) An example CAVEMEN-XML *affordance* node. Arrows inside the box point from parent nodes to child nodes. Arrows outside the specify different *formalism* attribute values and point to corresponding SAL model infrastructure. Nodes are colored to identify corresponding SAL code: in b–d, variable declarations are enclosed in color-coded rectangles and value assignments are written in color-coded text. (b) SAL model infrastructure generated for the *formalism* attribute value *stoffregen*. (c) SAL model infrastructure generated for the *formalism* attribute value *greeno*. (d) SAL model infrastructure generated for the *formalism* attribute value *chemero*

- The *affordance* node's *name* attribute followed by `[BOOLEAN, BOOLEAN]` in SAL (Fig. 6.4b, black, dashed-line rectangle)

Its value is (*true*, *true*) if the pair of Boolean variable values representing environment properties and human capabilities are true (Fig. 6.4b, red text). Other possible values could be (*true*, *false*), (*false*, *true*), or (*false*, *false*), depending on the respective values of local Boolean variables representing environment properties and human capabilities respectively.

When the *formalism* attribute is “greeno” (Fig. 6.4c), the translator generates model infrastructure corresponding to Greeno’s formalism (6.6) [6]. This includes one Boolean type output variable (Fig. 6.4c, black, dashed-line rectangle) representing the preconditions under which $\langle\langle$*action by agent* $\Rightarrow$ *good effects in situation*$\rangle\rangle$ (6.6) is satisfied. Its value (Fig. 6.4c, red text) is *true* if all conditions specified within *relation* nodes (Fig. 6.4c, blue text) as well as *translatable*, *positionable*, and *orientable* nodes (Fig. 6.4c, green text) are satisfied.

When the *formalism* attribute is “chemero” (Fig. 6.4d), the translator generates model infrastructure corresponding to Chemero’s formalism (6.7) [7]. This includes a function *affords* that takes two Boolean-values inputs and returns a Boolean-valued output if both inputs are *true*. The function corresponds to *Affords*-ϕ(*feature*, *ability*) in Chemero’s formalism (6.7). One input is a Boolean local type variable corresponding to a feature of the environment:

- *feature* in Chemero’s formalism (6.7)
- `feature_`, followed by the *affordance* node’s *name* attribute in SAL (Fig. 6.4d, blue dashed-line rectangle)

Its value is *true* if spatial relations in the HES satisfy those represented within CAVEMEN-XML *affordance* nodes (Fig. 6.4d, blue text). The translator also generates one Boolean type local variable corresponding to human operator abilities:

- *ability* in Chemero’s formalism (6.7)
- `ability_`, followed by the *ability* node’s *name* attribute in SAL (Fig. 6.4d, green rectangle)

Its value is *true* if human capabilities in the HES satisfy those represented within *translatable*, *positionable*, and *orientable* nodes (Fig. 6.4d, green text). A Boolean output variable is generated to capture the output of *affords*, and its name is the *affordance* node’s *name* attribute (Fig. 6.4d, black, dashed-line rectangle).

6.3.12 HES Modeling Technique

As mentioned, the analyst can add model infrastructure to the automatically generated HES module in order to specify static capabilities of the human operator(s) and evolving spatial relations among

HES entities. Initial spatial relations that are possible can be encoded using a technique similar to the one described in Chapter 5: the analyst could reason about what initial spatial relations are possible in a particular situation and encode them accordingly. This technique is useful for considering a constrained set of initial configurations (demonstrated in Section 6.4.4). Otherwise, the analyst can enable random initializations that are assigned during a model checking analysis. This technique is useful for considering a wide range of configurations within a single analysis, however it could include state variable assignments representing impossible configurations (such as small object surfaces covering larger ones).

Transitions should be encoded to specify what next-states of spatial relations are possible when each affordance exists and is actualized (discussed in greater detail below). This can be accomplished in many ways, depending on model complexity, architecture, and what affordance formalism is instantiated. A general encoding is shown below for a CAVEMEN-SAL model having m affordances instantiated using either Greeno's or Chemero's formalism (Fig. 6.4b, c), where *affordance_1 . . . affordance_n* are Boolean variables (encoding for Stoffregen's formalism shown in Appendix C.1).

```
TRANSITION [
 affordance_1 -->
  entity_1' = ...
       ...
  entity_n' = ...
 affordance_m -->
  entity_1' = ...
       ...
  entity_n' = ...
  ...
 []ELSE -->
];
```

This code specifies that if an affordance exists, one or more spatial relations can change in the next-state. A final "ELSE" guard command ensures that there are no deadlock states (i.e. states in which the model cannot transition).

Semantically, these transitions represent changes to spatial relations effected immediately after an affordance is actualized, which is an indirect way of encoding the effects of a human action without representing such actions explicitly (as done in Chapter 5). Such a technique adopts the

convention employed in all three extant formalisms specifiable in CAVEMEN-XML: a human operator actualizing an affordance effects changes in the human-environment system. What relations can change when an affordance is actualized depends on the model, and there are many ways to encode these changes. One such technique is demonstrated in Section 6.4.4.

Initial human capabilities are assigned using the SAL initialization construct, as they are static according to Gibson's definition, unless the human operator loses a motor capability at some point in the HES evolution [8]. Such a scenario is not considered on this work. For each output variable having the `abilities` record type, the analyst should consider how a human operator can manipulate an object independently of all other objects. If it is a whole object (i.e. it was generated from a *modelobject* node having prefix m or a standalone *atomicobject* node having prefix ao), the analyst should specify movements that are possible for the human operator when it is only the human and object in the operational environment. Otherwise, the analyst should consider movements that are possible when the object, its permanently attached child object(s), and the human are in the operational environment. There are many ways such initializations can be encoded, and one technique is demonstrated in Section 6.4.4. Further examples are provided in Appendix C.2.

6.3.13 Module Composition

The translator automatically generates a system model named `affordances` by asynchronously composing the automatically generated `affordance` and `HES` modules. The asynchronous composition abstracts human-system interaction by reflecting updates to spatial relations in the HES that emerge in response to a human action. The SAL syntax specifying this module composition is generated as shown below:

```
affordances:  MODULE = affordance [] HES;
```

6.3.14 Specifications

As mentioned, accuracy and error tolerance are important characteristics of safety-critical system hardware. With respect to the CAVEMEN modeling methodology, four kinds of LTL specifications were developed to assert these characteristics.

In regard to accuracy, configurable hardware must be designed to enable the correct affordances at the appropriate times. Therefore the analyst may need to verify that the temporal relationships among one or more affordances are correct. For example, if a cable output end is connectable to an input source, it may be necessary for a different output end to be connectable to a different input source in the future, where the first connectability affordance transitioning from available to unavailable ensures that the other connectability affordance remains available.

In this work, the LTL specification asserting such a situation is called *Positive affordance accuracy*, as it represents a positive (i.e. desired) temporal relationship between two safe affordances (i.e. two affordances that are both necessary to configure the system correctly). To encode such a specification, the CAVEMEN model must represent at least two affordances that should emerge at different times. The example specification in (6.10) reads, "it is always true (G) that when a safe affordance exists ($safeAffordance_i$) and ($\wedge$) does not exist in the future ($\mathsf{F}(\neg safeAffordance_i)$, i.e. some change occurred disabling it) this implies ($\Rightarrow$) that, in the future (F), a different safe affordance ($safeAffordance_j$) emerges."

$$\mathsf{G}(safeAffordance_i \wedge \mathsf{F}(\neg safeAffordance_i) \Rightarrow \mathsf{F}(safeAffordance_j)) \tag{6.10}$$

A different version of this specification can be encoded to assert the absence of a negative (i.e., undesired) temporal relationship between two affordances (i.e., actualizing one affordance ensures that an undesired affordance does not emerge in the future). For example, if a cable output end is disconnectable from a discharged battery, the end user should not be able to connect the cable to the same discharged battery in the future. In this work, such a specification is called *Negative affordance accuracy*, as it specifies a desired temporal relationship between a safe and unsafe affordance. The example specification in (6.11) reads, "it is always true (G) that when a safe affordance currently exists ($safeAffordance$) and ($\wedge$) does not exist in the future ($\mathsf{F}(\neg safeAffordance)$), this implies ($\Rightarrow$) that, in the future ($\mathsf{F}$), an unsafe affordance does not emerge ($\neg unafeAffordance$)."

$$\mathsf{G}(\mathit{safeAffordance} \wedge \mathsf{F}(\neg \mathit{safeAffordance}) \Rightarrow \mathsf{F}(\neg \mathit{unsafeAffordance})) \tag{6.11}$$

In regard to error tolerance, the analyst may need to verify that an unsafe affordance never emerges when the HES is in a particular configuration. Consider the aircraft fuselage door example discussed earlier in this chapter: a critical safety feature of the door is that it is not openable when the aircraft exceeds a certain altitude. In this work, such a specification is called *Weak affordance error tolerance*. The example specification in (6.12) reads, "it is never true ($\mathsf{G}\neg$) that when the HES model is in a particular configuration ($\mathit{variable} = \mathit{value}_i$), this implies ($\Rightarrow$) that an unsafe affordance does not emerge ($\neg \mathit{unsafeAffordance}$)."

$$\mathsf{G}\neg(\mathit{variable} = \mathit{value}_i \Rightarrow \neg \mathit{unsafeAffordance}) \tag{6.12}$$

Finally, the analyst may need to verify that a particular affordance never emerges in any HES configuration. For example, configurable hardware may need to be designed such that a cable output end is never connectable to the incorrect input socket. In this work, such a specification is called *Strong affordance error-tolerance*. The example specification in (6.13) reads, "it is always true (G) that an unsafe affordance never emerges ($\neg \mathit{unsafeAffordance}$)."

$$\mathsf{G}(\neg \mathit{unsafeAffordance}) \tag{6.13}$$

6.3.15 Model Checking Technique

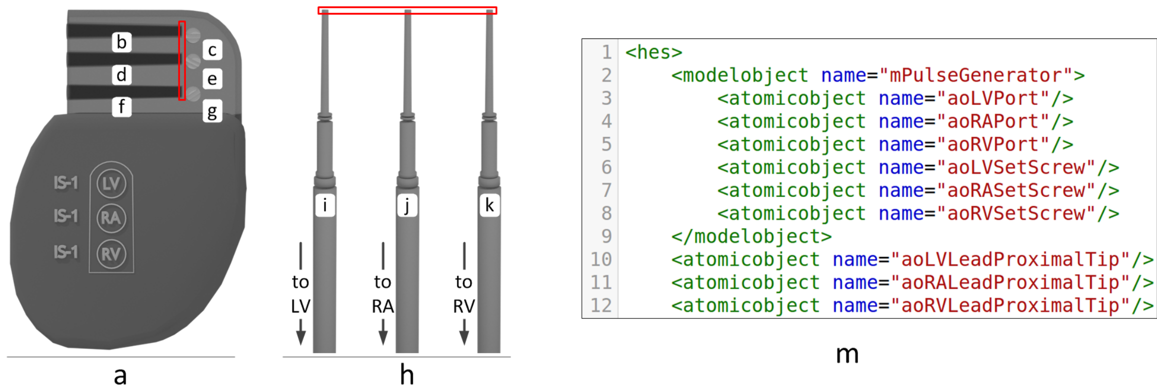
Specifications can be encoded manually in SAL using the theorem construct [68] (see Chapter 3, Section 3.2.3.1 for more information on the SAL theorem construct). They can be verified using either symbolic model checking (SAL-SMC) or bounded model checking (SAL-BMC) [68] (see Chapter 3, Section 3.2.3 for more information about these tools).

6.4 Case Study

To demonstrate the application of the CAVEMEN approach, a medical device adverse event involving the emergence of an unsafe affordance was selected from the U.S. FDA Manufacturer and User Facility Device Experience (MAUDE) database [203]. The MAUDE database contains searchable records of medical device adverse events reported by hospitals, outpatient centers and other user facilities. Reports include information such as an event description and a manufacturer narrative. The report was selected from a search for adverse events occurring in 2015 involving a "connection issue" and a patient injury. The search returned 195 results, and the 29th report was selected using a random number generator.

The selected case study involves a surgically implanted cardiac resynchronization therapy pacemaker (CRT-P) intended for use in heart failure patients (MAUDE report number 2124215-2015-13749): the manufacturer received a report indicating that a patient's CRT-P device was not providing optimal therapy. Testing and evaluation revealed that all three of the device's leads were connected to the wrong input ports on the pulse generator: the RA lead was connected to the LV port, the RV lead was connected to the RA port, and the LV lead was connected to the RV port. The patient underwent an additional surgery to correct the connections.

HES entities, an initial HES configuration, two safe affordances, and an unsafe affordance were identified using the MAUDE event description, manufacturer narrative, and information from the CRT-P device manufacturer's website. Safe affordances are specified as correct lead-to-port connectability for the LV and RV leads and ports, while the unsafe affordance is specified as LV lead connectability to the RV port. A CAVEMEN-XML specification of the HES and three affordances were encoded using Greeno's formalism [6] to specify the preconditions under which the affordances emerge. The instantiated CAVEMEN-XML representation was translated to SAL using the tool described in Section 6.3.9, and a separate model of the HES was encoded using the technique described in Section 6.3.12. A set of possible initial conditions were encoded by adding infrastructure to the automatically generated HES module. One set of initial conditions specifies motor capabilities of the surgeon enabling her to move all components of the CRT-P device along all six degrees of freedom.




```
1  <hes>
2      <modelobject name="mPulseGenerator">
3          <atomicobject name="aoLVPort"/>
4          <atomicobject name="aoRAPort"/>
5          <atomicobject name="aoRVPort"/>
6          <atomicobject name="aoLVSetScrew"/>
7          <atomicobject name="aoRASetScrew"/>
8          <atomicobject name="aoRVSetScrew"/>
9      </modelobject>
10     <atomicobject name="aoLVLeadProximalTip"/>
11     <atomicobject name="aoRALeadProximalTip"/>
12     <atomicobject name="aoRVLeadProximalTip"/>
```

Figure 6.5: (a–h) Graphical rendering of case study device components. Red rectangles indicate proximal tip and input port surfaces that must touch to establish a connection. (a) Pulse generator. (b) LV input port. (c) LV set screw. (d) RA input port. (e) RA set screw (f) RV input port. (e) RV set screw. (h) Lead segments. Arrows indicate continuation of middle segments and implanted positions of distal tips. Proximal tip segments begin at the top edges of white rectangles containing letters i–k. (i) LV lead proximal tip. (j) RA lead proximal tip. (k) RV lead proximal tip. (m) Lines 1–12 of instantiated CAVEMEN-XML model.

The second set of initial conditions specifies possible spatial relations among objects in the HES with respect to each other and the surgeon. Two guarded transitions specify spatial relations that emerge in the next-state following actualization of each affordance.

LTL affordance specification introduced in Section 6.3.14 were encoded and verified using symbolic model checking, and results are presented in Section 6.4.6.

6.4.1 System Description

A CRT-P device monitors heart functionality and delivers corrective electrical impulses to the patient's heart when potentially life-threatening cardiac abnormalities are detected. The system includes a programmed pulse generator having an internal battery and three leads. The tip of each lead (Fig. 6.5i–k) is 3.2 mm in diameter, fitting 3.48 mm diameter ports on the pulse generator [26]. The pulse generator (Fig. 6.5a) is 4.45 cm wide, 6.1 cm high, 0.75 cm deep and weighs 34 g. Its upper segment has three input ports of the same size and shape, circular with a 3.48 mm diameter (Fig. 6.5b, 6.5d, 6.5f). Each input port has a set screw for securing leads (Fig. 6.5c, 6.5e, 6.5g). Set screws cannot be removed from the pulse generator, but they must be loosened for leads to be inserted.

Leads (Fig. 6.5h) have distal and proximal ends with elongated, flexible middle segments. They are all of the same type having the same dimensions. During implantation a surgeon connects distal tips to three chambers of the patient's heart. Chambers include the left ventricle (LV), which pumps oxygenated blood throughout the patient's body, the right atrium (RA), which receives low-oxygen blood from the patient's body, and the right ventricle (RV), which receives low-oxygen blood from the RA and pumps it to the patient's lungs.

Distal tips of each lead are implanted at heart chambers indicated by arrows in Fig. 6.5h. Proximal tips (Fig. 6.5i–k) can be connected to the pulse generator via input ports. Based on the physician's technical manual [204], a lead is connected if complete electrical contact is made, i.e. the front surface of a proximal tip (Fig. 6.5h, areas enclosed within red rectangle) is covering the back surface of a port (Fig. 6.5a, areas enclosed within red rectangle). The appropriate lead–port connections are embossed on the lower segment of the pulse generator in Fig. 6.5a (vertical column of three circles labeled LV, RA, RV from top to bottom).

6.4.2 Event Description

One unsafe affordance that could be implicated in this case study is LV lead connectability to the RV port, which enabled an erroneous connection during implantation surgery: one action by the surgeon led to an unsafe connection established established in the next-state. Using Greeno's formalism, the action and its effects can be encoded as shown in (6.14).

$$\langle\langle \text{action by surgeon} \rangle\rangle \Rightarrow \langle\langle \text{LV lead connected to RV port} \rangle\rangle \tag{6.14}$$

Two safe actions and their effects, the LV and RV leads being connected to the correct ports, can be encoded in similar ways:

$$\langle\langle \text{action by surgeon} \rangle\rangle \Rightarrow \langle\langle \text{LV lead connected to LV port} \rangle\rangle \tag{6.15}$$

$$\langle\langle \text{action by surgeon} \rangle\rangle \Rightarrow \langle\langle \text{RV lead connected to RV port} \rangle\rangle \tag{6.16}$$

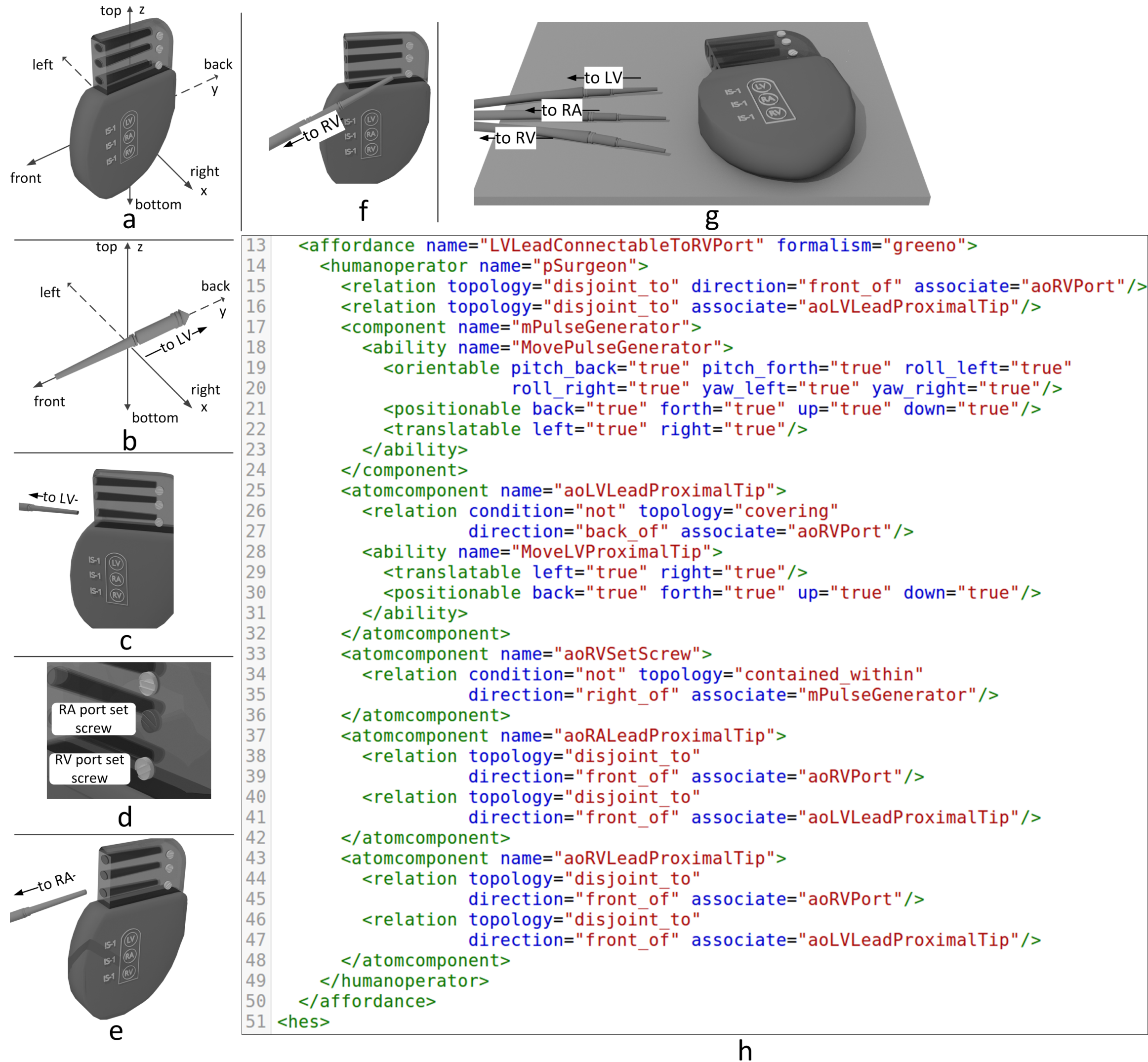


Figure 6.6: Graphical rendering of HES entities in a configuration satisfying *relation* nodes and attributes. Labeled axes in (a) show the surgeon's visual perspective with respect to surfaces of the pulse generator, all three ports, and all three set screws. Labeled axes in (b) show the surgeon's visual perspective with respect to all surfaces of the LV lead proximal tip. RA and RV leads are perceived in the same way. (a) Surgeon (not shown) disjoint to front of RV port. (b) Surgeon (not shown) disjoint to top, bottom and front of LV lead proximal tip. (c) LV lead proximal tip not covering back of RV port. (d) RV set screw touching right of pulse generator, and RA set screw disjoint to right of pulse generator for visual comparison. (e) RA lead proximal tip disjoint to front of RV port. (f) RV lead proximal tip disjoint to front of RV port. (g) RA lead proximal tip and RV lead proximal tip both disjoint to front of LV lead proximal tip (tabletop surface not modeled). (h) Lines 13–51 of CAVEMEN-XML model.

Greeno's formalism incorporates a verbal description of the preconditions enabling an affordance [6]. For this case study, they can be described as shown in outline form below:

1. The surgeon can move the pulse generator

2. The surgeon can move:

    (a) The LV lead proximal tip in support of (6.14) and (6.15)

    (b) The RV lead proximal tip in support of (6.16)

3. No HES entity is blocking the connection to:

    (a) The RV port in support of (6.14) and (6.16)

    (b) The LV port in support of (6.15)

4. The set screw is loosened for:

    (a) The RV port in support of (6.14 and (6.16)

    (b) The LV port in support of (6.15)

These preconditions are encoded formally in the next section using CAVEMEN-XML.

6.4.3 CAVEMEN-XML Model

In the specification description provided in text of this section, all node names, attribute names, and attribute values are italicized. The CAVEMEN-XML specification is shown in two graphical representations. Device components (Fig. 6.5a–h,) are specified in Fig. 6.5m. Preconditions of the unsafe affordance (6.14) are specified in Fig. 6.6h. The safe affordance preconditions (6.15) are not shown in Fig. 6.6). It was encoded by modifying the code in Fig. 6.6h by replacing each instance of "*aoRVPort*" with "*aoLVPort*" and "*aoRVSetScrew*" with "*aoLVSetScrew*" (discussed further in Section 6.4.3.2). The full model was 81 lines of XML code (Appendix F.1).

The root node *hes* is specified on line-1 of Fig. 6.5n. The pulse generator (Fig. 6.5a) is specified on lines 2–9 using a *modelobject* node with the *name* attribute valued *mPulseGenerator*. Ports (Fig. 6.5b, 6.5d, 6.5f) are specified on lines 3–5 using *atomicobject* nodes with *name* attributes *aoLVPort*, *aoRAPort*, and *aoRVPort* respectively. Set screws (Fig. 6.5c, 6.5e, 6.5g) are specified on lines 6–8 with *name* attributes *aoLVSetScrew*, *aoRASetScrew*, and *aoRVSetScrew* respectively.

All three lead proximal tips are specified within *atomicobject* nodes (Fig. 6.5i–k). The LV lead proximal tip (Fig. 6.5i) is specified on line-10 with the *name* attribute valued *aoLVLeadProximalTip*.

The RA lead proximal tip (Fig. 6.5j) and RV lead proximal tip (Fig. 6.5k) are specified on lines 11 and 12 respectively using the same syntax.

6.4.3.1 LVLeadConnectableToRVPort

The affordance of LV lead connectability to the RV port is specified on lines 13–54 of Fig. 6.6h. It is assigned the *name* attribute *LVLeadConnectableToRVPort*. The *formalism* attribute *greeno* indicates that the specification will employ the terminology and theory of Greeno's formalism [6].

While there are many HES configurations satisfying *relation* nodes specified in the remainder of this section, to aid the reader, one such configuration from Fig. 6.6a–g is referenced beside several *relation* node descriptions as (Fig. 6.6h, line number(s); Fig. 6.6 letter(s)).

The surgeon is specified using a *humanoperator* node with the *name* attribute *pSurgeon* (Fig. 6.6h, line-14). As mentioned, the surgeon cannot be blocking the connection. One way to specify this is using *relation* nodes to assert that there must be empty space between all surfaces of the surgeon (i.e. waist, palms, fingers) and the front-side surface of the RV port (Fig. 6.6h, lines 15–16; Fig. 6.6a). Similarly, the surgeon cannot be blocking the connection by interfering with any surfaces of the LV lead proximal tip. This is specified on line-17 of Fig. 6.6b.

The pulse generator is specified using a *component* node with the *name* attribute *mPulseGenerator* (Fig. 6.6h, line-18). The surgeon must be capable of moving the pulse generator in a way that supports connectability of the LV lead proximal tip. This is specified on using an *ability* node with the *name* attribute *MovePulseGenerator* (Fig. 6.6h, line-19). As discussed in Section 6.3.7, this motor capability is defined with respect the pulse generator's origin, without considering where the surgeon and objects in the HES are in relation to each other at any one moment (alternatively, considering all imaginable spatial relations in any situation). For example, the pulse generator and lead tips could be facing different directions, positioned at different heights, or be in some other configuration that requires the surgeon to execute complex maneuvers of both components simultaneously. All movements of the pulse generator along x, y, and z axes could therefore be necessary for the affordance *LVLeadConnectableToRVPort* to emerge. This is specified on lines 20–25 using *orientable*, *positionable*, *translatable* nodes with all attributes valued *true*. Spatial relations of

the pulse generator are not specified because none are considered within the verbal description of preconditions identified earlier.

The LV lead proximal tip is specified using an *atomcomponent* node with the *name* attribute *aoLVLeadProximalTip* (Fig. 6.6h, line-28). As mentioned, the affordance requires that the LV lead proximal tip is not already connected to any input port of the pulse generator (i.e., it is not covering the back-side surface of any port). This is specified on lines 29–34 of Fig. 6.6h. Additionally, the surgeon must be capable of moving the LV lead proximal tip in a way that supports insertion to the RV port in any situation. This is specified using an *ability* node with the *name* attribute *MoveLVLeadProximalTip* (Fig. 6.6h, line-35). As mentioned in the *MovePulseGenerator ability* node description, the surgeon may need to move both the pulse generator and the LV lead proximal tip along the x, y, and z axes simultaneously to actualize the affordance *LVLeadConnectableToRVPort*. However, since the lead can be connected to a pulse generator input port without rotating it, there are no situations in which the movements *roll_left* or *roll_right* are clearly necessary. This is specified on lines 36–40 using *orientable*, *positionable*, *translatable* nodes with all attributes valued *true*, except for *roll_left* or *roll_right*, which are omitted (i.e., they could be either *true* or *false* for the affordance to emerge).

The set screw for securing leads connected to the RV port is specified using an *atomcomponent* node with the *name* attribute *aoRVSetScrew* (Fig. 6.6h, line-43). As described in the system's physician technical manual [204], the set screw must be loosened for the LV lead proximal tip to be connectable. While this could be specified in many ways using CAVEMEN-XML *relation* nodes, images in the physician's technical manual suggest that the screw is tightened when no surfaces are touching the side of the pulse generator (in this case, the right side, as shown in Fig. 6.6a). Otherwise, if one or more surfaces are touching the right-side surface of the pulse generator, it is loosened (Fig. 6.6h, lines 44–45; Fig. 6.6d).

As mentioned, for the LV lead to be connectable to the RV port, no other lead proximal tips can be blocking the connection. These conditions are specified on lines 47–52 of Fig. 6.6h. On line-47, the RA lead is specified using an *atomcomponent* node with the *name* attribute *aoRALeadProximalTip*.

The *relation* node on line-48 specifies that it must be disjoint to all surface of the RV port. The same syntax is utilized for the RV lead proximal tip on lines 50–52.

6.4.3.2 LVLeadConnectableToLVPort

Because lead proximal tips and pulse generator input ports are all the same size and shape, the affordance of LV lead connectability to the LV port was specified using a modified version of the CAVEMEN-XML nodes and semantics described Section 6.4.3.1. This was accomplished in four steps:

1. Duplicating lines 13–54 of Fig. 6.6h
2. Replacing instances of "*RVPort*" on lines 13, 16, 48, and 51 with "*LVPort*"
3. On line-16, replacing the *associate* attribute valued "*aoRVPort*" with "*aoLVPort*"
4. On line-43, renaming the *atomcomponent* "*aoRVSetScrew*" to "*aoLVSetScrew*"

6.4.3.3 RVLeadConnectableToRVPort

As in Section 6.4.3.2, the affordance of RV lead connectability to the RV port was specified using a modified version of the CAVEMEN-XML nodes and semantics described Section 6.4.3.1. This was accomplished in two steps:

1. Duplicating lines 13–54 of Fig. 6.6h
2. Replacing instances of "*LVLeadProximalTip*" on lines 13, 17, 28, 35, and 50 with "*RVLead-ProximalTip*"

The final specifying all three affordances model was 138 lines of CAVEMEN-XML code.

6.4.4 SAL Representation

For the 138-line CAVEMEN-XML representation, the translator generated 115 lines of SAL code. In the `affordance` module, Boolean output variables named `LVLeadConnectableToRVPort`, `LVLeadConnectableToLVPort`, and `RVLeadConnectableToRVPort` are generated and valued using

equality assignments. Each variable is *true* if all respective spatial relation and human capability conditions are satisfied.

Utilizing the automatically generated HES module, infrastructure was added to specify initial motor capabilities of the surgeon and next-state spatial relations among HES entities. Assuming that the surgeon is able-bodied, abilities were initialized to allow pulse generator, LV lead proximal tip, and RV lead proximal tip movements along all six degrees of freedom (Appendix F.2, lines 84–89).

Initial spatial relations were encoded manually as a constrained set of possibilities, the majority of which were captured by removing configurations that are impossible:

- Components cannot be two places at once; e.g., if the LV lead proximal tip is covering the back surface of an input port, then it must be disjoint to all other components
- Components cannot be covering larger ones; e.g., the LV lead proximal tip cannot be covering the surgeon
- Due to 3.48 mm lead port diameters, the surgeon can only touch, cover, or overlap their front surfaces. She must be disjoint to all others
- Due to their size, lead proximal tips cannot contact opposite surfaces of an input port simultaneously; e.g., if the RA lead proximal tip is touching the right-side surface of the LV input port, is must be disjoint to the left-side surface

The SAL syntax specifying these initializations is shown in Appendix F.2, lines 84–141.

Next-state spatial relations were specified to represent changes emergent in the immediate next-state after one of three affordances are actualized. For these relations to emerge, guarded transitions specify that the corresponding affordance must be available. These guarded and next-state spatial relations are encoded in SAL as shown below (annotated in italic text).

```
TRANSITION [
 LVLeadConnectableToRVPort -->                             If the affordance is actualized,
  FORALL(d:  directional):                                 for all surfaces, the LV lead proximal tip is
    aoLVLeadProximalTip'.aoRVPort[d] = covering;           covering the RV port,
    aoLVLeadProximalTip'.aoLVPort[d] = disjoint;           disjoint to the LV port, and
    aoLVLeadProximalTip'.aoRAPort[d] = disjoint;           disjoint to the RA port
 LVLeadConnectableToLVPort -->                             If the affordance is actualized,
  FORALL(d:  directional):                                 for all surfaces, the LV lead proximal tip is
    aoLVLeadProximalTip'.aoLVPort[d] = covering;           covering the LV port,
    aoLVLeadProximalTip'.aoRVPort[d] = disjoint;           disjoint to the RV port, and
    aoLVLeadProximalTip'.aoRAPort[d] = disjoint;           disjoint to the RA port
 RVLeadConnectableToRVPort -->                             If the affordance is actualized,
  FORALL(d:  directional):                                 for all surfaces, the RV lead proximal tip is
    aoRVLeadProximalTip'.aoRVPort[d] = covering;           covering the RV port,
    aoRVLeadProximalTip'.aoLVPort[d] = disjoint;           disjoint to the LV port, and
    aoLVLeadProximalTip'.aoRAPort[d] = disjoint;           disjoint to the RA port
 []ELSE -->                                                Otherwise, nothing changes
];
```

Including these transitions, the final model was 206 lines of SAL code (Appendix F.2).

6.4.5 Specifications

One of each affordance-related usability specification was encoded to verify that configurable hardware is accurate and error tolerant:

1. *Positive affordance accuracy*: LV lead connectability to the LV port ensures that RV lead connectability to the RV port emerges in the future

This specification is needed to ensure that configurable hardware supports the surgeon in connecting the LV and RV leads to the correct input ports; more specifically, the first safe affordance of LV lead connectability to the LV port becoming unavailable in the future ensures that the second safe affordance of RV lead connectability to the RV does become available. This specification is encoded in SAL as shown below. It reads, "it is always true that if LV lead connectability to the LV port exists currently and does not exist in the future, RV lead connectability to the RV port emerges in the future."

```
PositiveAffordanceAccuracy:  THEOREM affordances |-
 G(LVLeadConnectableToLVPort AND F(NOT LVLeadConnectableToLVPort) =>
  F(RVLeadConnectableToRVPort));
```

2. *Negative affordance accuracy*: LV lead connectability to the LV port ensures that LV lead connectability to the RV port does not emerge in the future

This specification is needed to ensure that configurable hardware supports the surgeon in connecting the LV and RV leads to the correct input ports; more specifically, if the safe affordance of LV lead connectability to the LV port becomes unavailable in the future, then the unsafe affordance of LV lead connectability to the RV port should not emerge in the future. This specification is encoded in SAL as shown below. It reads, "it is always true that if LV lead connectability to the LV port exists currently and does not exist in the future, LV lead connectability to the RV port does not emerge in the future."

```
PositiveAffordanceAccuracy:  THEOREM affordances |-
 G(LVLeadConnectableToLVPort AND F(NOT LVLeadConnectableToLVPort) =>
  F(NOT LVLeadConnectableToRVPort));
```

3. *Weak affordance error tolerance*: LV lead connectability to the RV port never emerges when the system is in a particular configuration

This specification is needed to ensure that configurable hardware prevents the surgeon from connecting the LV lead to the RV port when the system is in a particular configuration. In this case study, one such configuration could be the one in which the RV lead has been configured correctly (i.e., it is connected to the RV port). This specification is encoded in SAL as shown below. It reads, "it is always true that when the RV lead is connected to the RV port, this implies that the LV lead is not connectable to the RV port."

```
WeakAffordanceErrorTolerance:  THEOREM affordances |-
 G(aoRVLeadProximalTip.aoRVPort[back_of] = covering =>
  NOT LVLeadConnectableToRVPort);
```

4. *Strong affordance error tolerance*: LV lead connectability to the RV port never emerges

This specification is needed to ensure that configurable hardware always prevents the surgeon from connecting the LV lead to the RV port. The SAL syntax encoded below, reads, "it is never true that LV lead is connectable to the RV port."

```
WeakAffordanceErrorTolerance:  THEOREM affordances |-
 G(NOT LVLeadConnectableToRVPort);
```

6.4.6 Verification

Specifications were verified using SAL's symbolic model checker (SAL-SMC) [68]. Results, number of states visited, and execution times are shown in Table 6.3.

Table 6.3: Case study model checking results

Specification name	Result	States visited	Verification time (s)
Negative affordance accuracy	*proved*	$4.9345040397182 \times 10^{22}$	3.92
Positive affordance accuracy	*counterexample*	$4.9345040397182 \times 10^{22}$	3.98
Weak affordance error tolerance	*proved*	$4.9345040397182 \times 10^{22}$	4.32
Strong affordance error tolerance	*counterexample*	$4.9345040397182 \times 10^{22}$	4.20

The verification report of *proved* for *Negative affordance accuracy* indicates that configurable hardware supports a safe situation: if LV lead connectability to the LV port (i.e., a desired affordance) is available in the initial state and becomes unavailable in the future, then LV lead connectability to the RV port (i.e., an unsafe affordance) will become unavailable in the future. A 3-step sequence showing one way of satisfying this specification could be:

1. LV lead to LV port and RV lead to RV port connectability affordances were available in the first-state (Fig. 6.7a)

2. RV lead to RV port connectability was actualized in the second-state (Fig. 6.7b)

3. LV lead to RV port connectability was unavailable in the third-state (Fig. 6.7c)

The counterexample to *Positive affordance accuracy* shows a 3-step trace through the model leading up to an unsafe situation (depicted in Fig. 6.8):

1. LV lead to LV port and RV lead to RV port connectability affordances were available in the first-state (Fig. 6.8a)

2. LV lead to RV port connectability was actualized in the second-state (Fig. 6.8b)

3. RV lead to RV port connectability was unavailable in the third-state (Fig. 6.8c)

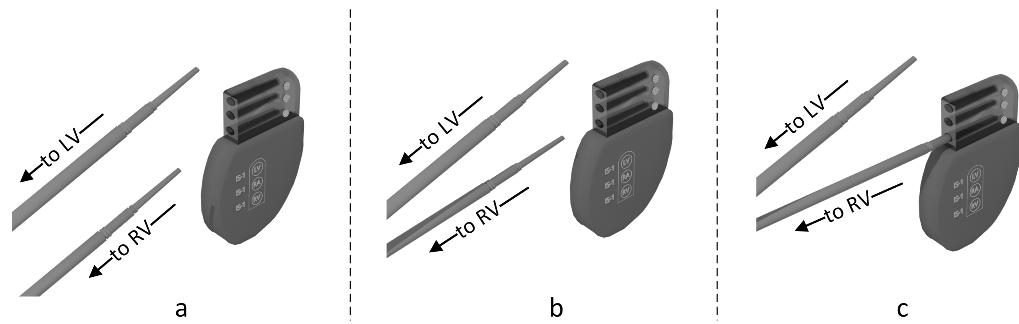


Figure 6.7: Rendering of the three-step trace leading up to a safe state in which the surgeon connects to RV lead to the RV port. (a) LV lead connectable to the LV port and the RV lead connectable to RV port. (b) RV lead connectability to the RV port is in the process of being actualized. (c) RV lead connected to the RV port

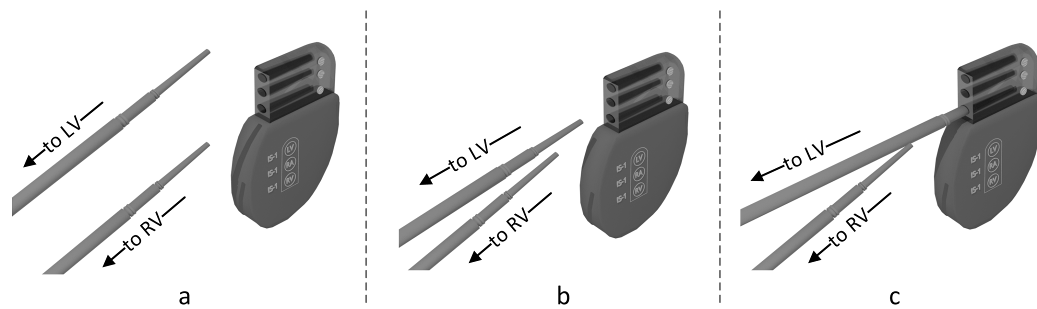


Figure 6.8: Rendering of the three-step trace leading up to an unsafe state in which the surgeon erroneously connects to LV lead to the RV port. (a) LV lead connectable to the LV port and RV lead connectable to the RV port. (b) LV lead connectability to the RV port is in the process of being actualized. (c) LV lead connected to RV port

Such a sequence reflects the surgeon erroneously connecting the LV lead to the RV port.

The verification report of *proved* for *Weak affordance error tolerance* indicates that configurable hardware supports a safe situation: when the RV lead is connected to the RV port, the LV lead is not connectable to the RV port. One configuration satisfying this safe state is depicted in Fig. 6.8c.

The counterexample to *Strong affordance error tolerance* shows a 0-step trace (i.e., the initial state) in which the unsafe affordance of LV lead connectability to the RV port is available. Two configurations representing this unsafe state are depicted in Fig. 6.8a and Fig. 6.7a.

6.5 Scalability

To evaluate scalability of the CAVEMEN approach, benchmark experiments were conducted by encoding a generic CAVEMEN-XML representation having one *affordance* node, one *humanoper-*

ator, one *relation* (*topology* attribute value randomly assigned, *direction* attribute omitted), one *atomcomponent* (corresponding to one *atomicobject*) and one *ability* node having all three child nodes with randomly assigned attribute values. The *affordance* node was duplicated and all *name* attributes were modified to make them unique. A *condition="not"* attribute was assigned to the duplicated *relation* node, and the CAVEMEN-XML representations were translated to SAL using the JavaScript-based tool. This creates a formal model having two unique affordances. The automatically generated HES module was modified by adding guarded transitions for each affordance enabling randomly-assigned spatial relations to emerge in the next-state. This SAL syntax was encoded as shown below for one transition in the HES module (annotations added in italic text).

```
TRANSITION [
 affordance_1 -->           If the affordance is available,
  p1' IN {x:  p1_rels |     next-state spatial relations of p1 are randomly assigned such that
   EXISTS(x:  directional, y:  topological):  p1.ao1[x] = y}; any of them could exist
```

6 increasingly larger formal models were generated by repeatedly duplicating CAVEMEN-XML translating them to SAL. SAL-SMC [68] was utilized to verify an LTL specification that is always true, forcing the model checker to enumerate all reachable states. This specification (6.17) is trivially always true due to repeated pairs of *affordance* nodes with opposing *relation* nodes (1 always incorporates *condition="not"* attribute).

$$\mathsf{G}\neg(affordance_1 \wedge affordance_2 \wedge \ldots \wedge affordance_{n-1} \wedge affordance_n) \tag{6.17}$$

The number of states visited and verification times for each model are reported in Table 6.4. Doubling the number of unique affordances in each model increased state space by S^2, where S is the number of states in the previous model. Verification times generally increased with model size, but decreased when the number of unique affordances was doubled from two to four and 32 to 64. Verification times remained below two seconds for all analyses.

Table 6.4: Results of scalability evaluation

Number of unique affordances	States visited	Verification time (s)
2	1.6777216×10^{7}	0.28
4	$2.81474976710660 \times 10^{14}$	0.26
8	$7.9228162514264 \times 10^{28}$	0.40
16	$6.2771017353867 \times 10^{57}$	1.07
32	$3.9402006196395 \times 10^{115}$	1.67
64	$1.5525180923007 \times 10^{231}$	1.42

6.6 Discussion

This chapter presented the CAVEMEN approach for applying Gibsonian affordance within model checking analyses of hardware configurability. The XML-based grammar and modeling technique provides a method for instantiating three extant formalisms from ecological psychology and specifying affordances that emerge for an end user. The formal semantics of CAVEMEN-XML enable the analyst to specify configurable hardware components, their part-whole compositions, spatial relations among entities within an environment, and, using the 6DoF standard [201], physical capabilities of the end user to move configurable hardware components. A JavaScript-based translation tool parses instantiated CAVEMEN-XML representations and generates formal models of HES entities and affordances in the syntax of SAL [68]. An accompanying HES modeling techniques provides a way of specifying end-user motor capabilities, possible initial spatial relations among entities in the environment, and next-state spatial relations that emerge after an affordance is actualized. A verification methodology provides two accuracy-related specifications and two error tolerance-related LTL specifications that support automated verification of hardware configurability via symbolic model checking.

To demonstrate an application of CAVEMEN-XML and the translation tool, Greeno's formalism was employed to specify three HES affordances based on a medical device adverse event:

1. One correct affordance enabling the surgeon to successfully connect the LV lead to the LV port

2. One correct affordance enabling the surgeon to successfully connect the RV lead to the RV port

3. One incorrect affordance enabling the surgeon to successfully connect the LV lead to the RV port

The translation tool was utilized to automatically to generate a formal model of these affordances and the hardware components involved. Manually encoded infrastructure specified static motor capabilities of a normally-abled surgeon, possible initial HES configurations, and next-states of spatial relations that emerge in the HES after each affordance is actualized. One of each hardware configurable specification was instantiated and verified using symbolic model checking. Case study results indicate that the CAVEMEN approach could be useful for ensuring as well as identifying potential problems with respect to accuracy and error tolerance of configurable hardware.

A second set of analyses was conducted to evaluate scalability of the approach using symbolic model checking on a 3.5 GHz workstation having 64 GB RAM. For this verification apparatus, scalability results indicate that the approach shows promise for enabling formal verification of hardware configurability in CAVEMEN-SAL models having up to 64 unique affordances and approximately 1.55×10^{231} states on the target workstation.

6.6.1 Methodological Considerations

The CAVEMEN approach is the first attempt of a formal modeling and verification methodology that applies Gibsonian affordance formalisms within model-based analyses of hardware configurability. The formal grammar of CAVEMEN-XML therefore employs the same terminology as a subset of existing formalisms; however, it does not support the analyst in instantiating all formalisms discussed in Section 6.1. For example, since the term *effectivity* is not incorporated within CAVEMEN-XML, an affordance cannot be specified using the same terminology utilized in [1, 9, 3, 115, 2] and [8]. Additionally, since an *affordance* node could contain many *ability* nodes, modifications enabling the analyst to instantiate all of the discussed formalism would require replacing *ability* with *effectivity* and constraining the grammar such that one *effectivity* child node can be encoded within

one *affordance* node. While these modifications could be incorporated within a different version of CAVEMEN-XML, it is unclear if new semantics would add practical value to the approach with respect to accuracy and error tolerance of configurable hardware.

6.6.2 Future Work

In regard to specifying affordance as directly perceivable properties, CAVEMEN-XML does not have semantics for specifying how affordances are perceived (i.e., through visual, audible or haptic sensory channels). Because *ability* nodes are constrained to motor capabilities, the language is constrained to kinesthetic perception of force required to move an object, regardless of what sensory function is utilized to perceive spatial relations among entities in the HES. Other researchers have represented different sensory channels explicitly within a Gibsonian affordance formalism [202], and it could beneficial to incorporate similar semantics in a future version of CAVEMEN-XML.

In regard to *modelobject*, *subobject*, and *atomicobject* nodes, the heterarchical-hierarchical encoding enabled by CAVEMEN-XML enables the analyst to specify part-whole relationships among objects within a human-environment system. However, CAVEMEN-XML does not have semantics for specifying dimensions (i.e. sizes) of these objects. Additionally, while the analyst can specify where entities are in relation to each other, there is no way of specifying dimensions of the environment or other conditions that could affect affordances, such as other human operators, lighting, humidity, and heat. Specification of object size and other environmental conditions could be useful for enabling more refined analyses; and while this level of detail could overwhelm a symbolic model checker, scalability results indicate that it could be feasible to incorporate more detail in models having up to 32 unique affordances on a workstation having 64 BG RAM. This should be explored in future work.

In regard to the verification methodology, two accuracy-related and two error tolerance-related specifications were developed in this work. Currently, it is possible for a *negative affordance accuracy* model checking result of *proved* to be interpreted as a false negative; i.e., in the case study, a correct interpretation of the verification report is that configurable hardware *supports* a safe situation (shown in Fig. 6.7). However, one possible sequence of states that could satisfy the specification

involves erroneously connecting the LV lead to the RV port; in this case, actualizing *LVLeadConnectableToRVPort* trivially causes it to become unavailable in the future. This constitutes an unsafe situation, and the analyst may misinterpret the verification report of *proved* as "proof" that such an unsafe situation is not possible. Thus, to support the analyst, it could be beneficial to refine the specification in future work so a result of *proved* can be interpreted as "proof" that configurable hardware not only supports a safe situation, but guarantees it.

Another area of future work concerns the applicability of additional specifications, such as time-efficiency of affordances that should emerge in a specified number of steps (e.g., if an affordance is actualized, a different affordance should emerge in the next state). One such specification is explored in Chapter 9.

Manually encoding initialization and transition infrastructure of the HES module constitutes a significant part of the CAVEMEN approach. Accomplishing this currently requires knowledge of SAL and, potentially, much cognitive effort for the analyst. The analyst must also be able to encode an affordance model in XML, encode specifications in LTL, and visualize model checking results as a 3-D rendering of hardware components. Future work should explore ways of facilitating these processes within a graphical development environment.

Chapter 7: A Formal Approach to Interface Interpretation: Modeling, Specification, and Verification of Signifiers[1]

The human-system interface, which includes procedures, displays, controls, configurable hardware, and information discoverable within accompanying documentation, needs to be understandable. To support understandability, designers often incorporate signifiers within the interface that are clues providing insights into the function, purpose and meaning of the system, component or widget [37]. In safety-critical systems, signifiers are needed to inform the end user about the current state of the system, what inputs are possible, what actions to take, as well as what the consequences of an action will be. Designers may consider cultural context, such as a red octagon pushbutton emulating a stop sign to signify "stop." They may also leverage legacy systems, such as digital phones that emulate the sound of a rotary phone's bell to signify "incoming call." Signifiers can operate through a visual channel, such shapes and colors; an audible channel, such as tones and volumes; and a haptic channel, such as textures and vibrations. To support completeness, signifiers can also operate through the documentation channel, such as text and diagrams within user manuals describing what shapes, tones, and vibrations mean.

For a visual, audible, or haptic property of an interactive system to operate as a signifier, an end user must be able to perceive it, identify it, and relate its identity to system information [206]. Consider a battery-powered system having an indicator light that could be green, yellow or colorless to signify charge levels of "full," "low," and "none" respectively. For a charge level of "low" to be signified, the device must have a light and control logic for illuminating the light yellow at the appropriate times, and the end user must be capable of perceiving color and identifying "yellow." If characteristics of the system and end user interact correctly, the signified meaning of "low" is considered an output of the human-system interface.

If an interface component has multiple perceivable properties, such as the different colors and blinking patterns of an indicator light, what is signified by one property could depend on another.

[1] An earlier version of Section 7.2.4 was published in [205]

In this work, the property on which another depends is considered linked. For example, a battery indicator light could have three identifiable colors: red, yellow, and green. A red light could have two identifiable patterns: solid to signify that the battery is nearly discharged and blinking to signify that it has malfunctioned. A pattern identified as "blinking" may depend on the color identified as "red" to signify a particular function or meaning, such as "battery malfunction" when the indicator light is blinking red; thus, what is signified by the indicator light's pattern is linked to its color.

In this work, signifiers could operate in a time-variant or time-invariant way. Time-variant signifiers are patterns having identities and signified information that depend explicitly on time [85], such as the temporal pattern of an audible alert emitted by a household smoke detector. A pattern identified as "chirp" emitted every few minutes could signify "low battery," while a continuous pattern identified as "wail" could signify "smoke detected." Either signified meaning could emerge after the pattern becomes identifiable to an end user [84]. Time-invariant signifiers have identities that do not depend explicitly on time, such as colors of an indicator light (e.g. "yellow," "red") and volume levels of an audible alert (e.g. "loud," "quiet").

Visual signifiers such as shapes, symbols, and labels may have different meanings depending on an end user's perspective, such as an arrow-shaped pushbutton that could be perceived as pointing up or down. Such a signifier is visual, time-invariant, and orientation-dependent. A visual signifier could operate in an orientation-dependent way if it has one or more asymmetrical halves/hemispheres [86] enabling an end user to identify its orientation. Consider a compass having an arrow shaped needle (asymmetrical along its horizontal axis). For an end user having sufficient vision capabilities and knowledge of compass functionality, the arrow operates in an orientation-dependent way to signify what direction the end user is facing, such as "facing east" when the end user identifies the arrow's orientation as "left."

Signifiers may operate as interface outputs through one or more channels to support different systems and end users. Depending on system and end-user capabilities, one signifier channel may be sufficient to convey the function, purpose or meaning of an interface component. Consider an interface having a digital display with numbers signifying the system's speed and an octagon light

that illuminates red to signify that the speed is zero. Only the visual channel could be necessary to signify speed, but all visual properties must operate consistently. For example, if the octagon light illuminates red while the digital display reads "0," the signifiers are consistent.

An interface may present the same system information redundantly using multiple signifiers operating through different channels concurrently. Consider a crosswalk in a busy city intersection. When it is safe to walk, there will often be a sign with the illuminated symbol of a person walking. There may also be a speaker box that periodically emits the words, "walk now." For the safety of end user groups with different hearing/vision capabilities and similar language capabilities, the audible and visual signifiers must be redundant; i.e., they must both signify "walk now" at the same time.

It may not be possible for designers to present all information about device functionality completely through visual, audible, and haptic signifiers. Such systems may require an additional signifier channel: printed or electronic documentation describing information that visual, audible, and haptic properties do not explicitly signify. Consider jumper cables for starting a vehicle with a discharged battery. A vehicle's user manual may have instructions indicating that the negative terminal of the source vehicle's battery should be connected to ground at the recipient vehicle; however, visual and kinesthetic properties of electrical/mechanical components involved in this task may not signify what is meant by "ground" or what parts of a vehicle function as ground. Signifiers can be called complete if the functions and meanings regarding the ground connection are discoverable within the user manual.

Signifiers are critical in the design of human-interactive systems, and problems can arise when they do not support usability. For example, in June 2015 a patient went into cardiac arrest during surgery, and a surgical team member reacted by pushing a red button labeled "stop" on the X-ray imaging system control panel. This action unexpectedly shut down the life support system, and the patient expired [207]. Visual signifiers on the button may have been insufficient for understanding its function, and technical documentation on the manufacturer's website does not explain what is meant by the red color and "stop" label [208]. This reflects a potential problem regarding signi-

fier completeness. In the Three Mile Island nuclear accident, human operators understood what was signified by individual indicator lights on the coolant system control panel; however, multiple, concurrently illuminated lights had conflicting meanings, which delayed the necessary interventions [209]. This reflects a potential problem regarding consistency of the interface. In domains that could require a human operator's persistent visual attention, such as ground transportation, aviation, and surgery, researchers have identified a need for signifiers that provide alerts and feedback redundantly through audible, visual, and haptic channels [210]. However problems emerge when multi-channel signifiers have conflicting meanings (i.e. they are not redundant) [211].

It would be useful to ensure early in the design cycle that signifiers are usable. While formal methods-based frameworks have proven useful for analyzing procedures, affordances, displays, controls, and actuators, they are currently limited with respect to signifiers. In this chapter, a new approach is developed to address this need. Minimal requirements of a formal signifier modeling methodology are listed in Section 7.1. A formalism, modeling technique, and encoding tool intending to meet these requirements are described in Section 7.2. Temporal logic specifications are developed in Section 7.2.3 to assert consistency, redundancy, and completeness of signifiers with respect to an instantiated formal signifier model. These specifications can be verified using either symbolic or bounded model checking. The approach is demonstrated in a medical device case study in Section 7.3, and scalability is evaluated in Section 7.4. Discussions of case study results, scalability evaluation results, methodological considerations, and future work follow.

7.1 Representing Signifiers Formally

In the spirit of formal methods, a signifier formalism is needed to support the development of models that are mathematically amenable to verification; and in support of applicability to human factors engineering problems, it would be beneficial for the formalism to leverage extant theories of human-system interaction. Section 7.1.1 lists the set of minimal requirements for such a formalism.

As in other formal methods-based frameworks, such as EOFM [10], an encoding tool could facilitate the model development process by providing structure, keywords, and a translation tool that generates the target syntax of a model checking system from an intermediate representation.

Section 7.1.2 lists the set of minimal requirements for such an encoding tool.

7.1.1 Requirements of a Signifier Formalism

1. The formalism should be capable of representing human-system interface components in their possible modes and configurations

In this research, the human-system interface includes displays, controls, configurable hardware, and accompanying documentation. Signifiers could be incorporated within all of these components, and what is signified could depend on their modes and configurations, such as:

- What is rendered on displays when the device is in a particular mode
- What cables are connected when the device is in a particular configuration
- What is explained in documentation with respect to each mode or configuration

Thus, the formalism should have semantics for representing interface components, including a way of identifying what component, mode, and configuration is being specified.

2. The formalism should be capable of representing what functions and meanings are signified by the human-system interface

As mentioned, signifiers are critical for informing end users about the function or meaning of a system, component, or widget [37]. Examples of functions include what a component does or what the consequences of acting on it will be, while examples of meanings include what state the device is in or what action(s) should be taken. The formalism should therefore enable the analyst to specify what functions and meanings can be signified.

3. The formalism should be capable of representing perceivable properties of human-system interface components

Components of a human-system interface, such as displays and widgets, could have one or more perceivable properties that operate as signifiers, such as text labels, audible alerts, and haptic vibrations. Researchers in HCI have identified that perceivable interface properties operating as

signifiers commonly belong to a particular component [212]; and part of what makes a property operate as a signifier is its composition within a higher-level element, such as text that is rendered within a dialog box, rather than within a different container on a visual display [213]. The formalism should therefore have semantics for representing what is signified by perceivable properties that belong to a component.

4. The formalism should be capable of representing what is signified to a representative end user by one or more perceivable properties

As mentioned, for a property to operate as a signifier, the end user must be able to perceive it, identify it, and relate its identity to a function or meaning [206]. Additionally, if multiple properties need to operate in parallel, such as the example of a blinking red light, the color and the visual pattern must be perceived, identified, and related to a signified function or meaning. Thus, the formalism should provide a way of abstracting these relationships. In formal methods, relationships between formalism elements are commonly represented as input/output functions. To enable such a representation, the signifier formalism requires two additional capabilities:

4.1 The formalism should be capable of representing sets of function inputs

A mathematically correct function needs to have a well-defined domain (i.e., a set of input values). With respect to the theorized characteristics of signifiers, function inputs should be one or more perceivable property identities assigned by the end user. Thus, because different perceivable properties could have different identity domains (e.g. "red" for a color input and "blinking" for a visual pattern input), the formalism should provide a way of representing distinct sets of property identities

4.2 The formalism should be capable of representing sets of function outputs

A mathematically correct function needs to have a well-defined (i.e., a set of outputs values). With respect to the theorized characteristics of signifiers in this research, function outputs should be one signified function or meaning. Because different perceivable properties of different interface components could signify different functions or meanings (e.g. "low" for the

charge level indicator light of a battery and "check charge level" for a pushbutton on the battery), the formalism should provide a way of representing distinct sets of signified functions and meanings

5. The formalism should be capable of representing signifiers that operate through different channels

In support of verifying signifier redundancy and completeness, the formalism needs to provide a way of representing what is signified to the end user concurrently through multiple channels. Thus, it should have semantics for representing distinct sets of signifiers that are grouped by channel, such as colors and symbols within a set of visual-channel signifiers; volumes and pitches within a set of audible-channel signifiers; textures and vibrations within a set of haptic-channel signifiers; and documented explanations of what is signified by properties on the device within a set of documentation-channel signifiers.

7.1.2 Requirements of an Encoding Tool

In formal methods, encoding tools facilitate the process of instantiating a formalism, such as custom languages and translators for generating model checking syntax from intermediate representations [10]. Considering the signifier formalism requirements listed in Section 7.1.1, this section lists minimal requirements for a tool facilitating its instantiation.

1. The language should enforce a hierarchical structure for representing interface components and their perceivable properties

In support of Requirements 1 and 3 of the signifier formalism, the language should be hierarchical. This necessitates infrastructure for representing interface components and their lower-level perceivable properties.

2. The language should provide infrastructure for representing sets of perceivable property identities

In support of Requirement 4.1 of the signifier formalism, the language should support the analyst in defining sets of perceivable property identities that encompass distinct input/output function

domains. To reduce the need for knowledge of formal methods, such infrastructure should provide a way of grouping these sets using natural language, such as using by assigning a name to identities corresponding to the property being identified (e.g. "depicted" for a visual symbol).

3. The language should provide infrastructure for representing sets of signified functions and meanings

In support of Requirement 4.2 of the signifier formalism, the language should support the analyst in defining sets of signified functions or meanings that encompass distinct input/output function codomains. To reduce the need for knowledge of formal methods, such infrastructure should provide a way of grouping these sets using natural language, such as by assigning names that aid in identifying categories of similar functions or meanings.

4. The language should provide keywords for representing a constrained set of perceivable properties that operate as signifiers

To support the analyst in instantiating the signifier formalism, it could be beneficial to provide a set of keywords representing a minimal set of perceivable properties operating as signifiers. Useful keywords could have names or prefixes that aid identifying the property; what channel it operates through; and whether it is time-variant, time-invariant, or orientation-dependent.

5. The language should support parsing capabilities

A translator could help facilitate the development of formal signifier models that are amenable to formal verification. It is therefore necessary that the language enables parsing capabilities, similar to the languages developed for other applications (e.g. EOFM-XML [10]). Parsing capabilities could also be useful for enabling automated generation of LTL specifications with respect to an instantiated model, similar to the tool described in [57].

7.2 The BIGSIS Approach

The BIGSIS approach has three elements:

1. A formalism for modeling signifiers, including a technique for instantiating it
2. An encoding tool, including a formal description language and translator
3. A verification methodology, including a set of LTL signifier specifications and a model checking technique

The BIGSIS formalism enables representations of signifiers and documentation as a formal model. Its semantics allow specification of what perceivable properties on a device signify functions and meanings, as well as what is signified by these properties based on explanations in the system's accompanying documentation. A modeling technique involves identifying sets of signified functions/meanings, properties operating as signifiers, components having perceivable properties, and documentation explaining what is signified. The analyst can assign initial end-user descriptions of perceivable properties, and for all subsequent states, next-state description assignments. An auxiliary modeling technique (not part of the BIGSIS formalism) provides a way of representing the system's control logic and/or human-system interaction controlling next-states of end-user descriptions. Formalism outputs represent one function or meaning for each category of functions/meanings signified through each channel (visual, audible, haptic, and/or documentation). For each category of signified function or meaning, one output value is randomly selected from a subset of model variables representing perceivable properties operating through each channel. This representation is useful for supporting model checking analyses that are nondeterministic, an approach that has proven useful in other formal methods-based analyses of human-interactive systems [61].

BIGSIS-XML is a custom, XML-based language for instantiating a formal signifier model, without the need for mathematical notation. Its custom grammar provides support for instantiating the BIGSIS formalism for a constrained set of visual, audible, and haptic properties operating as signifiers, as well as a system's accompanying documentation explaining what is signified. The language has infrastructure for representing categories of signified functions/meanings, what is signified by each property of each interface component, and what is signified by a property that depends on a different one (i.e., linked properties). The JavaScript-based translation tool parses an instantiated

BIGSIS-XML representation and generates a formal model adhering to the BIGSIS formalism.

For an instantiated BIGSIS-XML representation, the translator also generates a set of LTL signifier usability specifications that are verifiable using the SAL model checking system [68]. Three kinds of specifications are defined: consistency of single-channel signifiers, redundancy of multi-channel signifiers, and completeness of signifiers operating through a system's accompanying documentation. For each specification, a subset of specifications aids in conducting analyses with respect to a constrained set of signifier channels, a constrained set of device states, or a particular category of signified functions/meanings.

7.2.1 Representing Signifiers Formally: The BIGSIS Formalism and Modeling Technique

The BIGSIS formalism has semantics for specifying a formal signifier model based on an interactive system's electronic/mechanical components, documentation describing the system, end-user descriptions of visual, audible, and haptic properties (considered perceivable properties), and how descriptions relate to signified functions and meanings.

An interface component can be a display, cable, widget, or other electrical/mechanical element. Documentation can be a user manual, packaging label, or other form of printed or electronic documentation describing the system. Some examples of perceivable properties that can be specified using the BIGSIS formalism include:

- Time-invariant visual colors or labels
- Time-invariant, orientation-dependent shapes or symbols
- Time-invariant audible pitches or volumes
- Time-invariant haptic vibrations
- Time-variant audible, visual, or haptic patterns

An end-user description is a word or phrase that an end user would use to describe or identify a perceivable property, such as "red" to describe the color of indicator light. In determining end-user descriptions, the analyst should consider all visual, audible, and haptic properties presented by

electrical/mechanical interface components in their possible modes or configurations, such as when alarms are engaged, as well as an end user's relevant cognitive and perceptual capabilities. Similar descriptive words and phrases are grouped into named sets, such as red, yellow, and green in a set named "colors." Representing end-user descriptions in this way is useful for supporting the analyst in reasoning formally about end user characteristics independently of a system or device.

In determining functions and meanings that could be signified, the analyst should consider all interface components in their possible modes, all printed or electronic documentation accompanying the system, and the end user's characteristics such as cultural background and prior knowledge regarding system functionality. Functions and meanings are represented as words or phrases and separated into categories. A function word describes what the component does, how it can be acted upon by an end user, or what the consequence of an action will. A meaning word describes a state of the system such as current control mode. A category of function or meaning is a named, comma-separated list of words or phrases (e.g. "discharged," "low battery," and "charged" in a category of signified meanings named "charge levels"). Representing what is signified by category is useful for supporting model checking analyses of signifier consistency, redundancy, and completeness (discussed in Section 7.2.3).

To represent how functions and meanings are signified the analyst should consider all interface components having perceivable properties with end-user descriptions and any printed or electronic documentation accompanying the system. An interface component can be a display, cable, widget, or other electrical/mechanical element. The analyst should assign each component a name that helps to identify it. For example, if there is an alarm on a device controller that engages when the device is operating outside a threshold speed, the analyst could name it "speed alarm."

For each component, each visual, audible, and/or haptic property could signify one function or meaning, such as the red color of a "speed alarm" indicator light signifying "stopped." What is signified could also depend on another perceivable property, such as the blinking pattern of a red light. If accompanying documentation describes an interface component's perceivable properties, such as user manual text describing what an audible tone means, the analyst could specify this

information as operating through the documentation channel. The documentation channel represents functions and meanings signified by visual, audible, and haptic properties as described in text, lists, diagrams, and other printed or electronic materials accompanying the system.

Every visual, audible, and haptic property must be assigned an end-user description. Each end-user description defines how a perceivable property operates as a signifier, which could be in a time-variant or time-invariant way. Consider the example component named "speed alarm" having an indicator light that is illuminated red when the alarm is engaged. Considering the end user's perceptual capabilities such as ability to perceive and identify colors, the analyst could specify that one possible end-user description of color is "red." Considering the end user's characteristics such as culture and past experiences with other systems, the analyst could specify that the color description "red" relates to a signified speed of "stopped." The approach of assigning end-user descriptions and relating them to signified information is employed for two reasons:

1. It promotes formal reasoning about the user independently of the device or component, which could be useful for informing design considerations

2. It enables the analyst to model what changes occur as the device and the end user's perception of it evolve

In the example of a component named "speed alarm," "stopped" is signified when the end-user description of color is "red," corresponding to a particular mode: the alarm is engaged. Other end-user descriptions of color could also exist, such as "no color" when the alarm is not engaged, corresponding to a different mode and possibly relating to another speed signified elsewhere on the device, which the analyst could also specify.

A property having at least one end-user description operates as a signifier if a particular description relates to exactly one signified function or meaning from a particular category. Considering the speed alarm example, if other speeds are signified by the indicator light when its end-user description of color is "red" (i.e. there is a one-to-many relationship between description and signified meaning), it does not adhere to the formalism and it should not be specified. This convention is useful for discovering violations of signifier consistency (discussed in Section 7.2.3) during the formalism

instantiation process.

If a component has multiple properties of the same kind signifying the same category of function or meaning the analyst must specify them as properties of two separate components. This convention is useful for encoding specifications that utilize natural language to explicitly differentiate similar components and perceivable properties. For example, an application icon on a desktop PC screen could have two labels, one that is always present and one that appears when the user's cursor hovers over it. Both labels could signify what the application does. Instead of specifying the icon's labels as "label1" and "label2," the BIGSIS formalism requires the analyst to specify each as a "label" property of differently named components, such as "icon" and "icon hovered over," which is more descriptive than numbered labels.

Utilizing these semantics, the BIGSIS formalism specifies a formal model representing interface signifiers as a set of states, transitions, next-states, and outputs. The formal syntax of the BIGSIS formalism are described in the remainder of this section.

7.2.2 BIGSIS Formalism Semantics

The BIGSIS formalism represents a formal signifier model as a hierarchical composition of five Z [153] schemas (see Section 3.1 for information about the Z specification language):

1. *values*: specifies all signified functions and meanings, categories of common functions/meanings, and how an end user could describe properties of electrical/mechanical interface components

2. *properties_signify*: specifies visual, audible, and haptic interface properties having descriptions as well as functions/meanings that could be signified explicitly on the device and through information discoverable within accompanying documentation

3. *signifiers*: specifies variables representing interface components and accompanying documentation as well as initial descriptions and signified functions/meanings to all interface components and accompanying documentation

4. *next_state_signifiers*: specifies all subsequent next-state values of *signifiers* schema elements,

including optional model infrastructure representing the system's control logic and/or human-system interaction controlling next-state descriptions

5. *outputs*: specifies signifier channels and randomly selects one signified function or meaning output for each set of visual, audible, haptic, and documented functions or meanings

Variables of the *values* schema have basic types, and each subsequent schema has variables that are schema types. Variables of a schema type access variables from its type using a period. For example, if $S_1 : \{w_0, w_1, w_2\}$ is declared in the schema *values* and $P_1 : values$ is declared in a subsequent schema *properties_signify*, $P_1.S_1$ may appear within the *properties_signify* schema, and its value must be either w_0, w_1, or w_2.

Variables declared in a predicate are local to that predicate, and the same names can be reused in other predicates to represent distinct elements. For example, if one predicate in a schema contains a value d_0, e.g. $\exists_1 d_0 : D_1$, a different predicate in the same schema can also contain a variable d_0, e.g. $\exists_1 d_0 : D_2$, and the two instances of d_0 are distinct.

Within each schema, subscripts represent n distinct variables having the same elements in its type, such as $P_{1,\ldots,n} : values_{1,\ldots,n}$ representing n properties (each a variable) having corresponding schema types. The value of n is not specified, and its value could be different in each subscript (i.e., $P_{1,\ldots,n}$ and $S_{1,\ldots,n}$ does not mean that there are the same number of Ps and Ss). Subscripts referencing individual elements, such as i in P_i and j in P_j, indicate that P_i and P_j are distinct.

7.2.2.1 Values Schema

The *values* schema specifies all possible values in the model: end-user descriptions of perceivable properties as well as signified functions and meanings. Declaring all end-user descriptions and all signified functions and meanings in the first schema, as well as separating them into named lists, supports hierarchical modeling that is common in software engineering.

To identify end-user descriptions, the analyst should consider all visual, audible, and haptic properties of interface components, as well as the end user's relevant cognitive/perceptual capabilities and other characteristics that could shape interpretation such as cultural context and age. End-user

description words are listed within a set ([*descriptions*]). If there are multiple kinds of descriptions (e.g. colors, labels), common words go in a corresponding subset (D_i : *descriptions*) containing a set of words ($d_1, \ldots, d_n$).To identify signified functions and meanings, the analyst should consider device components, perceivable properties, and documentation describing the system.

All words are listed within a set ([*signified*]). Words that represent similar functions and meanings can be grouped into categories (S_i : *signified*) containing a set of words ($w_1, \ldots w_n$), each of which is named to identify what is common. The *values* schema also specifies one or more words describing perceivable properties of the interface.

values

$S_{1,\ldots,n}$: *signified*
$D_{1,\ldots,n}$: *descriptions*

$\bigcap_{i=1}^{n} S_i = \emptyset$

$\bigcap_{i=1}^{n} D_i = \emptyset$

$\forall\, category : S_i \bullet \exists_1 w_0 \in category$

$\forall\, descriptions : D_i \bullet \exists_1 d_0 \in descriptions$

The first two predicates specify that there can be no common elements among categories of signified function or meaning and no common elements among sets of descriptions respectively. The third predicate specifies that for all categories ($\forall\, category : S_i$), there should be one unique word in each one representing nothing signified ($\exists_1 w_0 \in category$, e.g. *alarmNotSignified* in a category named *Alarms*). Similarly the fourth predicate specifies that for all sets of end-user description words ($\forall\, descriptions : D_i$), there should be one unique word in each set representing the absence of a description ($\exists_1 d_0 \in descriptions$, *noColor* in a set of descriptions named *Colors*).

7.2.2.2 Properties_signify Schema

The *properties_signify* schema specifies perceivable properties operating as signifiers of function and meaning audibly, visually, haptically, and through accompanying documentation. Perceivable properties ($P_1, \ldots, P_n$) have the schema type *values*. Each property includes one set of end-user descriptions (D_i) having a set of descriptive words ($d_0, \ldots, d_n$), and one or more categories of signified function or meaning ($S_{1,\ldots,n}$), each having a set of function or meaning words ($w_0, \ldots, w_n$).

The declaration "P_1 : *values* $\setminus D_{2,\dots,n}$" specifies that the perceivable property declared as P_1 has one set of descriptions, D_1, and all other sets of descriptions are not included in its type. This is the case for all perceivable properties, up to P_n, which excludes all sets of end-user description words except for D_{n-1} (denoted by P_n : *values* $\setminus D_{1,\dots,n-1}$). Each P_i represents all instances of a perceivable property within the interface. For example, if there are many colors presented on a device and/or described within accompanying documentation, one P_i represents all colors and functions/meanings they could signify.

properties_signify

$P_1 : values \setminus D_{2,\dots,n}$

$\dots$

$P_n : values \setminus D_{1,\dots,n-1}$

$\bigcap_{i=1}^{n} P_j.S_i = \emptyset$

The predicate specifies that there are no common function or meaning words ($w_0, \dots, w_n$) among the sets $P_i.S_1$ through $P_i.S_n$; i.e., for all perceivable property variables, each can contain, at most, one of each function or meaning category.

7.2.2.3 Signifiers Schema

The *signifiers* schema specifies interface components ($C_{1,\dots,n}$) and accompanying documentation ($Doc_{1,\dots,n}$) describing components. The types of each component (C_i) and accompanying documentation (Doc_i) are instances of the *properties_signify* schema ($properties_signify_{1,\dots,n}$).

The set of functions $relation_{1,\dots,n} : D_1 \twoheadrightarrow S_1$ represents each relationship an end user could associate between one word from a set of description words in a particular group (D_i) and one word from a particular category of function or meaning (S_i). For each component, there is a set of relation functions for each of its perceivable properties and the categories of function or meaning those properties signify. For example, if an interface has n components, m perceivable properties, and o categories of function or meaning, then there can be as many as p relation functions, where $p = n \times m \times o$. A different set of functions ($explanation_{1,\dots,n} : D_i \twoheadrightarrow S_i$) represents accompanying documentation explaining what is signified. Similarly, if an interface has n components, m perceiv-

able properties, and o categories of function or meaning, all of which are explained in accompanying documentation, then there can be as many as p explanation functions, where $p = n \times m \times o$.

The analyst may need to encode this many unique relation and explanation functions for three reasons:

1. As specified in the *values* schema, every description type ($D_1, \ldots, D_n$) has a different set of elements ($d_0, \ldots, d_n$), which operates as the domain of a relation or explanation function. In formal methods, a function's domain must be well defined for it to be mathematically correct. Thus, a different relation (or explanation) function is needed for each description type (D_i)

2. Similarly, every category of function or meaning ($S_1, \ldots, S_n$) has a different set of elements ($w_0, \ldots, w_n$), which operates as the codomain of a relation or explanation function; thus, a different function is needed for each category type (S_i)

3. Two different components (or two different sets of documented explanations) could have the same property and the same end-user description, but a different signified function or meaning. While they could have the same domain and codomain, their output values could be unique, which makes the functions mathematically distinct. For example, a traffic light (i.e., an interface component) could be colored green to signify "safe to go straight" (encoded as *safeToGoStraight*). It could also have a separate, arrow-shaped light that is colored green at the same time to signify "safe to turn left" (encoded as *safeToTurnLeft*). These are two different components having the same property and, potentially, the same end-user description of "green" at the same time. Thus, two unique relation functions are needed to specify different output values (*safeToGoStraight* and *safeToTurnLeft*) for the same input value (*green*)

$$
\begin{array}{l}
\textit{signifiers} \\
\hline
C_{1,\ldots,n} : \textit{properties_signify}_{1,\ldots,n} \\
Doc_{1,\ldots,n} : \textit{properties_signify}_{1,\ldots,n} \\
\textit{relation}_{1,\ldots,n} : D_i \twoheadrightarrow S_i \\
\textit{explanation}_{1,\ldots,n} : D_i \twoheadrightarrow S_i \\
\hline
\forall\, c_i : C_i \bullet c_i.P_i.D_i = d_i \\
\forall\, c_i : C_i;\ c_j : C_j \bullet \\
\quad \textbf{if } \textit{relation}_{c_i.P_i.D_i}(c_i.P_i.D_i) \neq w_0 \\
\quad\quad \textbf{then } c_i.P_i.S_i = r(c_i.P_i.D_i) \vee \\
\quad\quad\quad c_i.P_j.S_i = c_i.P_j.S_i \vee \\
\quad\quad\quad c_i.P_j.S_i = c_j.P_{i \vee j}.S_i \\
\quad \textbf{else } c_i.P_i.S_i = w_0 \\
\forall\, c_i : C_i;\ doc_i : Doc_i;\ doc_j : Doc_j \bullet \\
\quad \textbf{if } \textit{explanation}_{c_i.P_i.D_i}(c_i.P_i.D_i) \neq w_0 \\
\quad\quad \textbf{then } doc_i.P_i.S_i = \textit{explanation}_{c_i.P_i.D_i}(C_i.P_i.D_i) \vee \\
\quad\quad\quad doc_i.P_i.S_i = doc_i.P_j.S_i \vee \\
\quad\quad\quad doc_i.P_i.S_i = doc_j.P_{i \vee j}.S_i \\
\quad \textbf{else } doc_i.P_j.S_i = w_0 \\
\hline
\end{array}
$$

The first predicate (starting with $\forall\, c_i : C_i \bullet$) specifies that for all components having a set of perceivable properties, each property's description ($c_i.P_i.D_i$) is assigned a descriptive word (d_i). This includes the possibility of $d_i = d_0$ (i.e., the word corresponding to no description, such as *noColor*). The second predicate (starting with $\forall\, c_i : C_i;\ c_j : C_j : C$) specifies what is signified by interface component properties without accompanying documentation. A conditional expression states:

1. If the relation function output for the component and its property ($\textit{relation}_{c_i.P_i.D_i}(c_i.P_i.D_i)$) is not equal ($\neq$) to the word representing nothing signified (w_0), then what is signified can be one of three things:

   (a) a function or meaning word from the category S_i, produced by the relation function ($\textit{realtion}_{c_i.P_i.D_i}(c_i.P_i.D_i)$),

   (b) a linked perceivable property, which must be a different perceivable property if it comes from the same component ($c_i.P_j.S_i$), or

   (c) any linked perceivable property from a different component ($c_j.P_{i \vee j}.S_i$).

Otherwise, nothing is signified (**else** $c_i.P_i.S_i = w_0$). The "**else**" case includes descriptions having

one-to-many relationships, such as a red color of an indicator light simultaneously signifying device states of "stopped" and "malfunction."

The third predicate (starting with $\forall\, c_i : C_i;\ doc_i : Doc_i;\ doc_j : Doc_j$) specifies what is signified by properties of interface components as they are explained within accompanying documentation. It is equivalent to the third predicate, replacing each instance of c_i that comes after "$explanation_{c_i.P_i.D_i}(c_i.P_i.D_i)$" with "$doc_i$." What is signified through the documentation channel is determined by perceivable property descriptions of interface components; thus, variables representing documented descriptions ($doc_i.P_i.D_i$) are not utilized in the formalism.

As a visual aid, one relation and one explanation function are shown in Fig. 7.1 for a hypothetical smoke detector interface. The smoke detector has three audible alarms, one of which can engage when the battery is low, smoke is detected, or carbon monoxide is detected. For each alarm, a different audible pattern (encoded as *aPattern*) is emitted. For a hypothetical end user, four audible pattern descriptions (*chirp*, *wail*, *longBeep*, and *noPattern*) signify the meaning of the alarm through audible (Fig. 7.1a) and documentation (Fig. 7.1b) channels. In Fig. 7.1a, the relation function specifies that when the end user describes the audible pattern as "chirp" (*SmokeDetector.aPattern.Pattern* = *chirp*), the signified meaning is *lowBattery* (corresponding to $relation_{c_i.P_i.D_i}(c_i.P_i.D_i) = w_i$) when $c_i.P_i.D_i = d_i$). Similarly, the signified meaning for *SmokeDetector.aPattern.Pattern* = *wail* is *smokeDetected* (corresponding to $relation_{c_i.P_i.D_i}(c_i.P_i.D_i) = w_j$) when $c_i.P_i.D_i = d_j$). However, when the end user describes the audible pattern as either *longBeep* (discussed in the next paragraph) or *noPattern*, the meaning of the alarm is not signified through the audible channel (respectively corresponding to $relation_{c_i.P_i.D_i}(c_i.P_i.D_i) = w_0$) when $c_i.P_i.D_i = d_k$ and $relation_{c_i.P_i.D_i}(c_i.P_i.D_i) = w_0$) when $c_i.P_i.D_i = d_0$).

In Fig. 7.1b, the explanation function for accompanying documentation provides the same signified meanings for end-user descriptions of *chirp* and *wail*. However, in this example, text within accompanying documentation explains that a periodic, 1-second beep (for which the end-user description is *longBeep*) indicates that carbon monoxide is detected. Thus, the explanation function of the smoke detector's audible pattern determines a signified meaning of *carbonMonoxideDetected*

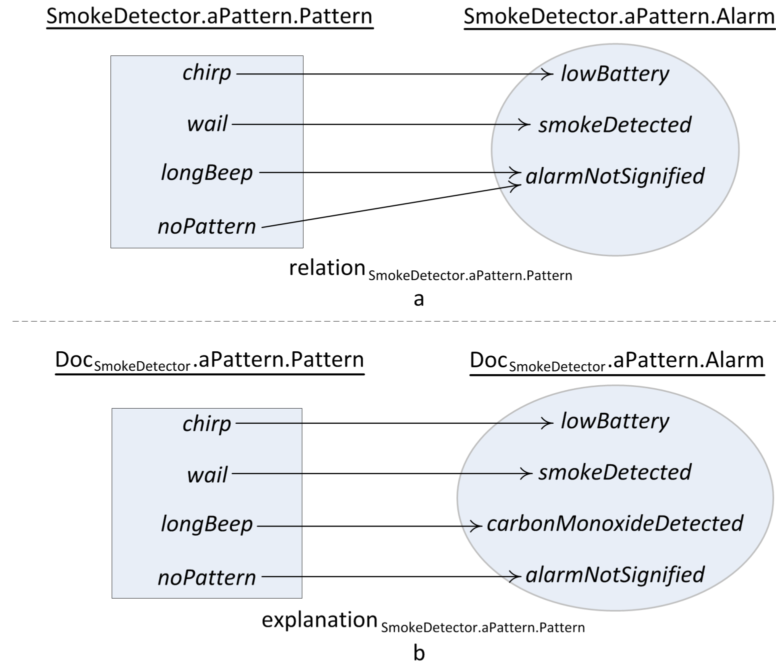


Figure 7.1: Visual representations of one relation function and one explanation function for a hypothetical smoke detector interface (a) and its accompanying documentation (b). Underlined text above each oval identifies *signifiers* schema variables instantiated for the smoke detector. Italic words within rectangles are a set of end-user description words from a set of audible pattern descriptions (corresponding to $\mathbb{F}_1 D_i$)). Italic words within ovals are a set of meaning words from a category of signified alarm meanings (corresponding to $\mathbb{F}_1 S_i$)). Arrows pointing from one description word to one meaning word represent the input/output behavior of each function (a) The relation function for the smoke detector's audible pattern. (b) The explanation function for accompanying documentation explaining what is signified by the smoke detector's audible pattern

for an end-user description of *longBeep* (corresponding to $explanation_{c_i.P_i.D_i}(c_i.P_i.D_i) = w_k$) when $c_i.P_i.D_i = d_k$).

7.2.2.4 Next_state_signifiers Schema

The *next_state_signifiers* schema coordinates changes to values assigned in the *signifiers* schema. Its declaration is $\Delta signifiers$, which introduces the current set of states in *signifiers* and another set of next-states (*signifiers'*). The formalism enables random next-states of end-user descriptions in all subsequent states.

$$
\begin{array}{|l}
\hline
next_state_signifiers \\
\Delta signifiers \\
\hline
\end{array}
$$

Optionally, the analyst could encode predicates in the *next_state_signifiers* schema and/or schemas considered separate from the BIGSIS formalism representing the system's control logic and human-system interaction in a way that controls next-states of end-user descriptions. One way to do this is by encoding a *device* schema (not part of the formalism) having outputs that are inputs to the *next_state_signifiers* schema. For example, consider the following *device* schema representing a household smoke detector:

$$
\begin{array}{l}
\textit{device} \\
\hline
\textit{Smoke} : \mathbb{N} \\
\textit{Alarm} : \{\textit{On}, \textit{Off}\} \\
\hline
\textit{Smoke} \geq 5 \Rightarrow \textit{Alarm} = \textit{On} \\
\textit{Smoke} < 5 \Rightarrow \textit{Alarm} = \textit{Off} \\
\hline
\end{array}
$$

The *next_state_signifiers* schema of a corresponding BIGSIS formalism instantiation could utilize *Alarm* as an input in a way that controls end-user descriptions of the alarm:

$$
\begin{array}{l}
\textit{next_state_signifiers} \\
\hline
\Delta \textit{signifiers} \\
\textit{Alarm} : \{\textit{On}, \textit{Off}\} \\
\hline
\textit{Alarm}' = \textit{On} \Rightarrow \textit{Detector}'.\textit{Volume}.\textit{Level} = \textit{Loud} \\
\textit{Alarm}' = \textit{Off} \Rightarrow \textit{Detector}'.\textit{Volume}.\textit{Level} = \textit{noLevel} \\
\hline
\end{array}
$$

In this example, the first predicate reads, "a next-state of $\textit{Alarm} = \textit{On}$ implies ($\Rightarrow$) that the next-state end-user description of the alarm's volume level ($\textit{Detector}'.\textit{Volume}.\textit{Level}$) is *Loud*." The second predicate reads, "a next-state of $\textit{Alarm} = \textit{Off}$ implies that the next-state end-user description of the alarm's volume level is *noLevel*." This technique is demonstrated in Section 7.3, represented graphically in Fig. 7.6.

7.2.2.5 Outputs Schema

The *outputs* schema specifies functions and meanings signified by the interface for each category and each channel. Its types are outputs (denoted by !), corresponding to categories signified through up to four channels: visual, audible, haptic, and documentation ($\textit{signified}!_{1,\dots,n}$). The value of each output must come from a corresponding category of signified function or meaning, $S_{1,\dots,n}$.

outputs

$$signified!_{1,\ldots,n} : S_{1,\ldots,n}$$

$$\forall\, visually_signified_i : signified_i! \bullet$$
$$visually_signified_i = w_i \wedge w_i \in C.vis.S_i \vee$$
$$C.vis.S_i = \emptyset \Rightarrow visually_signified_i = w_0 \wedge w_0 \in S_i$$

$$\forall\, audibly_signified!_i : signified_i! \bullet$$
$$audibly_signified_i = w_i \wedge w_i \in C.aud.S_i \vee$$
$$C.aud.S_i = \emptyset \Rightarrow audibly_signified_i = w_0 \wedge w_0 \in S_i$$

$$\forall\, haptically_signified_i : signified_i! \bullet$$
$$haptically_signified_i = w_i \wedge w_i \in C.hap.S_i \vee$$
$$C.hap.S_i = \emptyset \Rightarrow haptically_signified!_i = w_0 \wedge w_0 \in S_i$$

$$\forall\, documented_i : signified!_i;\ \bullet$$
$$documented_i = w_i \wedge w_i \in Doc.P.S_i \vee$$
$$Doc.P.S_i = \emptyset \Rightarrow documented_i = w_0 \wedge w_0 \in S_i$$

The first three predicates represent what is signified through visual, audible, and haptic channels. If there are m channels and n categories, there are $m \times n$ output variables. Properties operating through each respective channel are grouped into three basic types ([*vis*], [*aud*], [*hap*]). The set of all electrical/mechanical components ($\bigcup_{i=1}^{n} C_i$) are represented as C, accompanying documentation ($\bigcup_{i=1}^{n} D_i$) as *Doc*, and properties ($\bigcup_{i=1}^{n} P_i$) as P.

The first predicate states that each output variable representing a visually signified function or meaning ($visually_signified_i$) is assigned a value (w_i) selected randomly from the set of visual properties ($w_i \in C.vis.S_i$). Otherwise, if there are no components having visual properties signifying the specified category ($C.vis.S_i = \emptyset$), this implies ($\Rightarrow$) that nothing is signified ($visually_signified_i = w_0 \wedge w_0 \in S_i$). The second and third predicates specify the same conditions for audible and haptic channels respectively.

The fourth predicate represents what is signified through the documentation channel. It is equivalent to the first three predicates, except a function or meaning word (w_i) could come from the set of all perceivable properties for which documentation describes what is signified ($w_i \in Doc.P.S_i$). If there is no such documentation ($Doc.P.S_i = \emptyset$), this implies ($\Rightarrow$) that nothing is signified through the documentation channel ($documented_i = w_0 \wedge w_0 \in S_i$).

Random selection for output values enables all possible ways a function or meaning could be signified through a particular channel; i.e., it is possible for an end user to utilize information

provided by any signifier at any time, and the probability of one signifier taking precedence over another is not modeled.

7.2.3 Specifications

To support formal verification of BIGSIS formalism models, three LTL specifications were developed. As discussed in Chapter 1, a critical aspect of interface understandability is that signifiers operating through the same channel do not have conflicting meanings. This characteristic is asserted in a signifier consistency specification (discussed in Section 7.2.3.1). Another critical characteristic for understandability is that signifiers operating through multiple channels concurrently do not have conflicting meanings. This is asserted in a signifier redundancy specification (discussed in Section 7.2.3.2). Finally, if signifiers presented on the device are insufficient for supporting end user understanding, such as a numeric error code or beeping audible alarm, the meanings of such signifiers must be explained in accompanying documentation. This characteristic is asserted in a signifier completeness specification (discussed in Section 7.2.3.3).

7.2.3.1 Signifier Consistency

Signifiers present on electrical/mechanical interface components are consistent if they operate through the same channel (audible, visual, haptic, or documentation) and always signify the same function or meaning. In a formal signifier model representing one or more electrical/mechanical interface components, one or more channels, and one or more categories of signified function or meaning, *signifier consistency* is specified for all components, one channel (visual, audible, haptic, or documented), and one or more categories. If a channel does not have two or more perceivable properties, it cannot have a *signifier consistency* specification. Because there are four specifiable channels in BIGSIS-XML, there can be no more than four signifier consistency specifications for any instantiated formal signifier model: *visual consistency*, *audible consistency*, *haptic sconsistency*, and *documentation consistency*.

In (7.1), *vis*, represent sets of all perceivable properties operating through the visual channel; $C_1 \ldots C_n$ represent all interface components having visual-channel signifiers; and $S_1 \ldots S_m$ repre-

sent all categories of signified function or meaning. The specification reads, "all signifiers operating through the visual channel always signify the same function or meaning from the same category." Signifier consistency specifications for audible and haptic channels ar encoded in the same way, replacing the set of visual-channel properties with audible- and haptic-channel properties respectively.

$$\mathsf{G}\begin{pmatrix} C_1.vis.S_1 = C_2.vis.S_1 \wedge \ldots \wedge C_{n-1}.vis.S_1 = C_n.vis.S_1 \wedge \\ C_1.vis.S_2 = C_2.vis.S_2 \wedge \ldots \wedge C_{n-1}.vis.S_2 = C_n.vis.S_2 \wedge \\ \ldots \\ C_1.vis.S_m = C_2.vis.S_m \wedge \ldots \wedge C_{n-1}.vis.S_m = C_n.vis.S_m \end{pmatrix} \tag{7.1}$$

The documentation signifier consistency specification is encoded as shown in (7.2). Here, $Doc_1 \ldots Doc_n$ represent all interface components having documented explanations of what is signified; P represents the set of all perceivable properties; and $S_1 \ldots S_m$ represent all categories of signified function or meaning. The specification reads, "all signifiers operating through the documentation channel always signify the same function or meaning from the same category."

$$\mathsf{G}\begin{pmatrix} Doc_1.P.S_1 = Doc_2.P.S_1 \wedge \ldots \wedge Doc_{n-1}.P.S_1 = Doc_n.P.S_1 \wedge \\ Doc_1.P.S_2 = Doc_2.P.S_2 \wedge \ldots \wedge Doc_{n-1}.P.S_2 = Doc_n.P.S_2 \wedge \\ \ldots \\ Doc_1.P.S_m = Doc_2.P.S_m \wedge \ldots \wedge Doc_{n-1}.P.S_m = Doc_n.P.S_m \end{pmatrix} \tag{7.2}$$

7.2.3.2 Signifier Redundancy

Signifiers are redundant if they operate through different channels and always signify the same function or meaning. In a formal signifier model representing one or more categories of signified function or meaning and a set of signifier channel outputs for each category, signifier redundancy is specified for two or more channels and one category. If only one channel is represented, there can be no signifier redundancy specifications.

Incorporating all channels through which a category of function or meaning could be signified (up to four: visual, audible, haptic, and documentation) constitutes a *total signifier redundancy* specification. In a formal signifier model representing two or more channels and n categories of function or meaning, the maximum number of *total signifier redundancy* specifications is n. The example in (7.3) reads, "Audible, visual, haptic, and documentation channels always signify the same function or meaning."

$$\mathsf{G}\left(\begin{array}{l} visually_signified_i = audibly_signified_i \wedge \\ audibly_signified_i = haptically_signified_i \wedge \\ haptically_signified_i = documented_i \end{array}\right) \tag{7.3}$$

If a formal signifier model represents two channels, all signifier redundancy specifications are total. However if three or four channels are represented the analyst could incorporate them within *partial signifier redundancy* specifications. In a formal signifier model representing three channels and n categories of function or meaning, there can be a maximum of $3n$ two-channel redundancy specifications. One such specification could be visual and audible signifier redundancy (7.4), which reads, "Visual and audible channels always signify the same function or meaning."

$$\mathsf{G}(visually_signified_i = audibly_signified_i) \tag{7.4}$$

In a formal signifier model representing all four channels and n categories of function or meaning, there can be a maximum of $6n$ two-channel partial specifications and $4n$ three-channel partial specifications ($10n$ altogether). One such three-channel specification (7.5) could incorporate visual, audible, and haptic channels for a particular category of signified function or meaning. This specification reads, "Visual, audible, and haptic channels always signify the same function or meaning."

$$\mathsf{G}\left(\begin{array}{l} visually_signified_i = audibly_signified_i \wedge \\ audibly_signified_i = haptically_signified_i \end{array}\right) \tag{7.5}$$

7.2.3.3 Signifier Completeness

Signifiers are complete if a function or meaning from a particular category is signified by perceivable properties of the device; or, if not, it is discoverable through accompanying documentation. In a formal signifier model representing one or more categories of signified function or meaning, one or more perceivable channels (audible, visual, and/or haptic), and a documentation channel, *signifier completeness* is specified for one category, up to three perceivable channels, and the documentation channel. If a particular category is not signified through the documentation channel, there cannot be a *signifier completeness* specification for that category; and for a formal signifier model having n categories. the maximum number of completeness specifications is n. The example in (7.6) reads, “It is never true that a function or meaning is not signified audibly, visually, or through accompanying documentation.”

$$\mathsf{G}\neg\left(visually_signified_i = w_0 \wedge audibly_signified_i = w_0 \wedge documented_i = w_0\right) \tag{7.6}$$

7.2.3.4 Constrained Specifications

If the analyst encodes separate model infrastructure representing human-system interaction or the system’s control logic (such as the *device* schema demonstrated in Section 7.2.2.4), a specification could utilize device model variables to constrain model checking analyses in a way that only considered device states of interest. For example, the signifier redundancy specification in (7.7) incorporates a variable *Alarm* from a *device* schema representing the system’s control logic for alarms. This constrains model checking analyses to states in which an alarm is engaged; i.e., “when an alarm is engaged, the audibly and visually signified meanings of the alarm are always the same.”

$$\mathsf{G}\left(Alarm = On \Rightarrow visually_signified_i = audibly_signified_i\right) \tag{7.7}$$

7.2.4 BIGSIS-XML

Formal models and LTL specifications instantiated in Z are not amenable to model checking analyses, which are critical to the BIGSIS approach. Utilizing an XML-based approach similar to those described in Chapters 5 and 6, BIGSIS-XML enables such analyses. Its formal semantics are leveraged from the Z-based BIGSIS formalism (Section 7.2.2), and its contents are specified using the XSD standard [159].

The BIGSIS-XML grammar is represented graphically in Fig. 7.2. The root node is named *bigsis*. Its direct children are named *signified-functions*, *signified-meanings*, *signifier-properties*, and *property-documentation*. Direct children of *signifier-properties* and *property-documentation* are perceivable properties operating as signifiers.

7.2.4.1 Signified Functions and Meanings

The *signified-functions* and *signified-meanings* nodes (Fig. 7.2b) represent categories of information signified by interface components. The encoding process is similar to that of categories from the BIGSIS formalism *values* schema ($S_{1,\dots n}$). What is signified in each category could come from displays, controls, alarms, cables and any other components or widgets that have visually, audibly, or haptically perceivable properties as well as documentation describing what is signified. The analyst should first identify all functions and meanings that the interface (including documentation) can signify, and then represent each specific function or meaning as a word or phrase. As in the BIGSIS formalism, a function word describes what the component does, how it can be acted upon by an end user, or what the consequence of an action will be. A meaning word describes a state of the system such as a battery's charge level or the current alarm mode. Words or phrases that represent similar functions and meanings can be grouped into categories. The analyst can assign the *name* attribute a value that helps to identify what the similar words and phrases have in common.

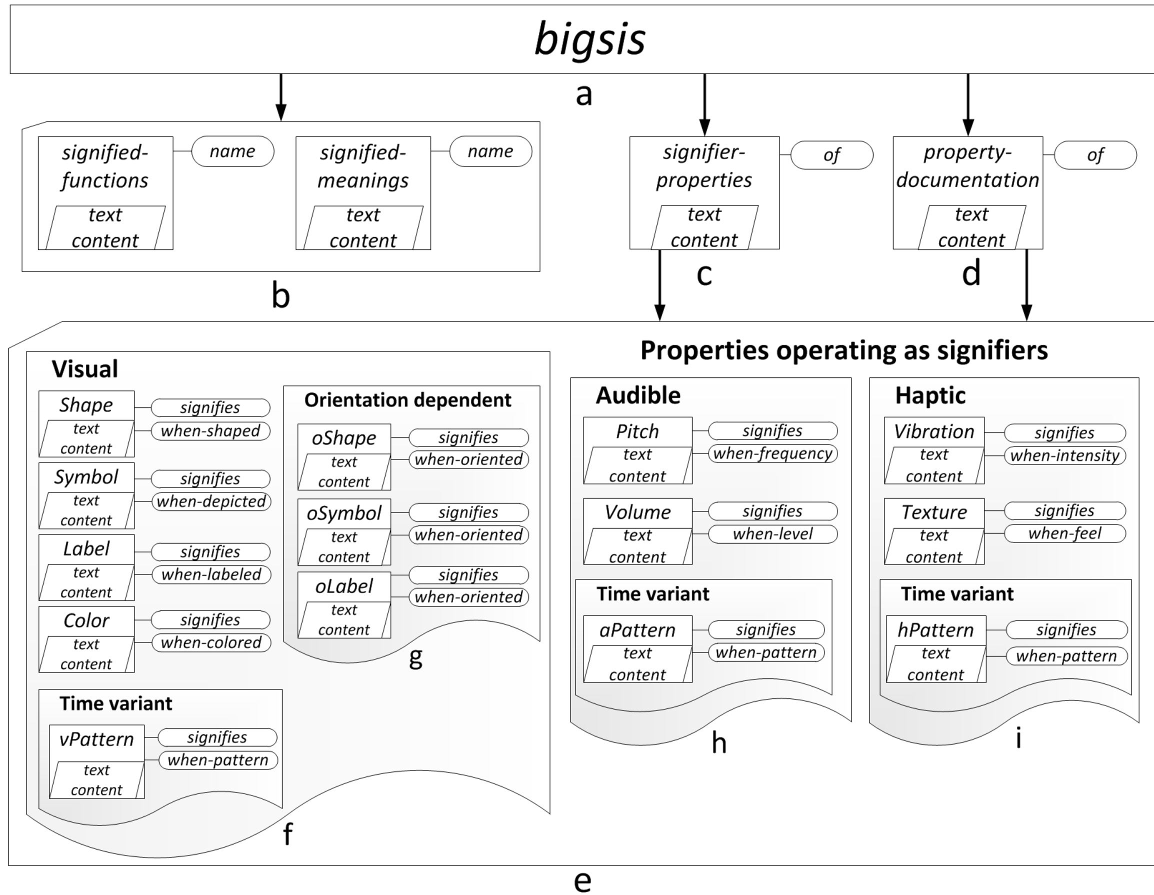


Figure 7.2: Visual representation of the BIGSIS-XML grammar. Square-edge rectangles are nodes. Smaller, round-edge rectangles are attributes. Parallelograms are text content. Arrows point from parent to child nodes. Boldface headings aid in identifying groups of similar perceivable properties and are not part of the grammar (a) The root node *bigsis*. (b–d) Direct children of *bigsis*. (e) Direct children of *signifier-properties* and *property-documentation*. The formal semantics of nodes f–g are leveraged from the BIGSIS formalism. (f) Visual signifiers. (g) Orientation-dependent visual signifiers. (h) Audible signifiers. (i) Haptic signifiers

Words are encoded as the text content of each node, separated by commas.

7.2.4.2 Signifier Properties

The *signifier-properties* (Fig. 7.2c) node allows the analyst to represent a mechanical and/or electronic interface component having one or more visually, audibly, and/or haptically perceivable properties. As in the BIGSIS formalism, an interface component can be a display, cable, widget, or other electrical/mechanical element. Each *signifier-properties* node must be assigned an *of* attribute, which should help to identify the component or section (e.g. display screen foreground) using a

unique, descriptive name (i.e. *of* attribute values cannot be repeated). Direct child nodes are one or more perceivable properties operating as signifiers (Fig. 7.2e).

7.2.4.3 Property Documentation

The *property-documentation* node (Fig. 7.2d) allows the analyst to represent all information about the device or system's components, modes or configurations provided within any form of printed or electronic documentation. As in the BIGSIS formalism, documentation may include user manuals, handbooks, manufacturers' websites, or other text-based sources.

The *property-documentation* node is encoded using the same syntax as the *signifier-properties* node. The *of* attribute for each *property-documentation* node must reference a component specified within an existing *signifier-properties* node. For example, if the analyst has encoded a *signifier-properties* node having the *of* attribute "speed alarm," a *property-documentation* node specifying documentation of the speed alarm must also be encoded using the same *of* attribute value. As in *signifier-properties* nodes, direct children are one or more perceivable properties operating as signifiers (Fig. 7.2e).

7.2.4.4 Properties Operating as Signifiers

For each *signifier-properties* and *property-documentation* node, the analyst can encode one or more child nodes representing a constrained set of perceivable properties operating as signifiers (Fig. 7.2e), referred to as perceivable property nodes. Throughout this section, the graphical representation in Fig. 7.3 aids in describing a constrained set of visual, audible, and haptic properties that are specifiable in the current implementation of BIGSIS-XML.

The analyst can specify visual properties of shape, symbol, label, and pattern. Visual patterns such as the blinking of an indicator light are considered time-variant, and they should be specified using prefix v for "visual, time-variant." All other properties are considered time-invariant and should be specified without a subscript, with the exception of orientation-dependent linked properties (discussed later). A perceivable property should be specified as a visual signifier if:

1. it is a perceivable property of shape, symbol, color, label, or pattern (Fig. 7.3a),

2. it has at least one definable, one-to-one relationship between its description and a signified function, meaning, or linked perceivable property.

If the first constraint does not hold, the property should not be specified. If the second constraint does not hold, the signifier may be poorly designed. This second constraint applies to all perceivable properties discussed throughout this chapter. If what is signified by the shape, symbol, or label depends on orientation from the end user's perspective, the analyst could link its signified function or meaning to an orientation-dependent perceivable property.

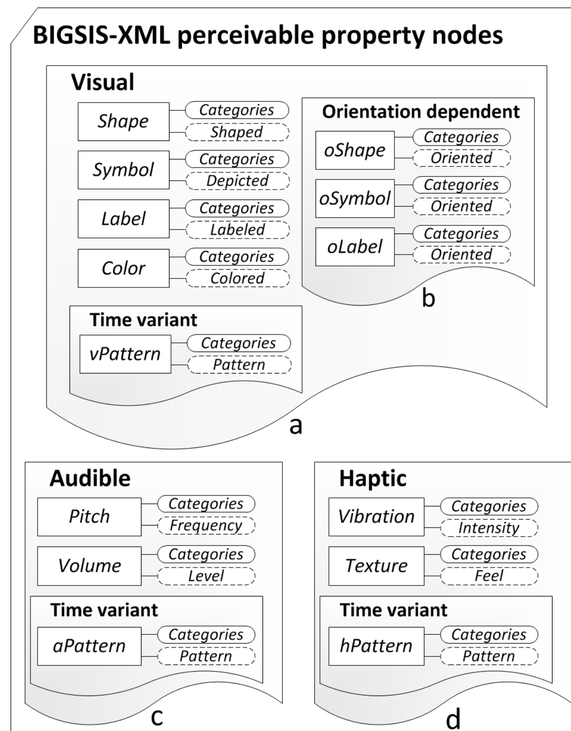


Figure 7.3: Graphical representation of signifier properties considered in the BIGSIS formalism. Boldface headings describe what is common among properties operating as signifiers. Signifier names are italicized within rectangles. Signifiers have one description (italicized within rounded, dashed edge rectangle) and one or more categories of signified function or meaning (italicized within rounded, solid edge rectangle) Prefix o stands for orientation-dependent. For time-variant properties of pattern operating through different channels, prefix v stands for visual, a for audible and h for haptic. (a) Visual signifiers. (b) Orientation dependent visual signifiers. (c) Audible signifiers. (d) Haptic signifiers

Shapes, symbols, and labels that have one or more asymmetrical axes could operate as signifiers in an orientation-dependent way, and their signified functions/meanings could be linked to orientation-dependent signifiers. A perceivable property should be specified as an orientation-dependent signifier

Table 7.1: Constrained set of BIGSIS-XML perceivable property nodes (corresponding to P_i of the BIGSIS formalism). Exemplars of descriptions (corresponding to D_i of the BIGSIS formalism) and descriptive words, terms, or phrases (corresponding to $d_0, \ldots, d_n$ of the BIGSIS formalism) are shown in the second and third columns

Perceivable property node (P_i)	Description (D_i)	end-user description words ($d_0, \ldots, d_n$)
Shape	*Shaped*	*noShape*, *triangle*, *octagon*, *square*
Symbol	*Depicted*	*noSymbol*, *arrow*, *checkMark*, *hourGlass*
Label	*Labeled*	*noLabel*, *stop*, *go*, *one*, *two*
Color	*Colored*	*noColor*, *red*, *yellow*, *green*
vPattern	*Pattern*	*noPattern*, *fastBlinking*, *slowBlinking*
oShape	*Oriented*	*noOrientation*, *up*, *down*
oSymbol	*Oriented*	*noOrientation*, *right*, *left*
oLabel	*Oriented*	*noOrientation*, *toward*, *away*
Pitch	*Frequency*	*noPitch*, *low*, *high*
Volume	*Level*	*noVolume*, *quiet*, *loud*
aPattern	*Pattern*	*noPattern*, *chirp*, *wail*
Vibration	*Intensity*	*noIntensity*, *low*, *moderate*, *high*
Texture	*Feel*	*noTexture*, *smooth*, *concave*, *notched*
hPattern	*Pattern*	*noPattern*, *rapid*, *slow*, *continuous*

using prefix o for "orientation-dependent" (Fig. 7.3b) if:

1. the analyst has already specified a shape, symbol, or label that could be linked to an orientation-dependent signifier,

2. the shape, symbol, or label has one or more definable orientations in two and/or three-dimensional space, and

3. it has at least one identifiable one-to-one relationship between a description of its orientation and a signified function, meaning, or linked perceivable property.

If the first constraint does not hold the analyst should specify a corresponding perceivable property without prefix "o" (*Shape*, *Symbol*, *Label*). If the second constraint does not hold the signifier may not operate in an orientation-dependent way.

The analyst can specify audible properties of pitch, volume, and pattern and haptic properties

of texture, vibration, and pattern. Audible patterns should be specified using prefix *a* for "audible, time-variant" and haptic patterns should be specified using prefix *h* for "haptic, time-variant." A perceivable property should be specified as an audible or haptic signifier if:

1. it is an audible property of pitch, volume, or pattern (Fig. 7.3c) or a haptic property of texture, vibration or pattern (Fig. 7.3d) and
2. it has at least one definable, one-to-one relationship between its description and a signified function, meaning, or linked perceivable property.

If the first constraint does not hold the property should not be specified.

For each node of Fig. 7.2e, the *signifies* attribute specifies what category of function or meaning the perceivable property signifies. Its value must be the *name* attribute of a *signified-functions* or *signified-meanings* node.

All perceivable property child nodes of *property-documentation* must have the same name and *signifies* attribute as a corresponding *signifier-properties* child node; i.e. the analyst cannot specify documentation of a perceivable property that does not operate as a signifier.

For each perceivable property node, the *when-* attribute (Fig. 7.2e, directly below *signifies* attributes) specifies an end-user description, e.g. *when-colored="red"* for a *Color* node. Text content specifies what is signified, i.e. the one-to-one relationship between an end-user description and a signified function, meaning, or linked perceivable property having the same *signifies* attribute.

The formal semantics of linked perceivable properties are the same in BIGSIS-XML and the BIGSIS formalism, and they are encoded using a *properties-signify* or *property-documentation of* attribute, a signifier property node, and a *signifies* attribute. Examples could be *Battery.Symbol.ChargeLevel* and *PushButton.oShape.SpeedAdjustment*. If the parent node is *signifier-properties*, a linked perceivable property always references a *signifier-properties* child node. Likewise, if the parent node is *property-documentation*, a linked perceivable property references a *property-documentation* child node.

As in the BIGSIS formalism, while encoding these nodes the analyst should consider the end user's perceptual and cognitive capabilities such as ability to perceive and identify colors as well

as end user characteristics such as culture and past experiences. This approach promotes formal reasoning about the user independently of the device or component, which could be useful for informing design considerations. Similarly, each node's *when-* attribute must relate to exactly one signified function or meaning from the same category or exactly one linked property, and a one-to-many relationship between an end-user description and a signified function, meaning, or linked property is not specifiable. The BIGSIS-XML grammar enforces this by restricting whitespace and commas within perceivable property node text content. These restrictions are useful for two reasons:

1. They could support the analyst in uncovering violations of signifier consistency during the encoding process. If a perceivable property could signify multiple functions or meanings, then it does not always signify the same thing as other perceivable properties.

2. Forcing the analyst to encode one-to-one relationships supports BIGSIS-XML representations that adhere to BIGSIS formalism semantics, specifically the *relation* and *explained* functions-defined in the *signifiers* schema (Section 7.2.2.3). This enables translation of BIGSIS-XML representations to model checking syntax, discussed next.

7.2.5 Formal Model Translation

The custom, JavaScript-based tool incorporated within the BIGSIS approach parses instantiated BIGSIS-XML representations and generates formal models and signifier specifications in the model checking syntax of SAL [68].

Translated BIGSIS-SAL models are constructed from enumerated, record types, a single module, and one or more theorems, depending on what LTL specifications are relevant (automatically generated by the translator). The module has local and output variables, initializations, transitions, and definitions. The following sections reference a visual representation (Fig. 7.4a) of a BIGSIS-XML instantiation. Node names, attributes, and text content of Fig. 7.4 were not derived from an interface; rather, they generic and serve to demonstrate how the translator parses BIGSIS-XML nodes and generates SAL syntax corresponding to formalism schemas.

7.2.5.1 SAL Representation of *values* Schema

To represent declarations of the *values* schema, the translator generates enumerated types for each *signified-functions* and *signified-meanings* node of a given BIGSIS-XML instantiation. Each type corresponds to a signified function/meaning category $S_i : \mathbb{F}_1 \mathit{signified}_i$ of the *values* schema. One such type is represented in Fig. 7.4c (`S1`). The set of values within each type is the node's text content and an additional, automatically generated word representing nothing signified (`S1NotSignified` in Fig. 7.4c, corresponding to a w_0).

Enumerated types are also generated to represent lists of end-user descriptions corresponding to $D_i : \mathbb{F}_1 \mathit{descriptions}_i$. The translator parses all perceivable property nodes that are direct children of *signifier-properties* (e.g. *Color*, *oSymbol*) and places their *when-* attribute values within enumerated types. One enumerated type is generated for each set of *when-* attributes having the same name. For example, if there are multiple *when-colored* attributes appearing in the BIGSIS-XML instantiation, their values are placed within an enumerated type named `COLORS`. The translator ensures there is a descriptive word within each type representing an absence of description (e.g. `noColor`, corresponding to d_0). Since *when-depicted*, *when-colored*, *when-pattern*, and *when-level* attributes appear in Fig. 7.4, enumerated types of `SYMBOLS`, `COLORS`, `PATTERNS`, and `LEVELS`, and are represented in Fig. 7.4a.

7.2.5.2 SAL Representation of *properties_signify* Schema

To represent declarations of the *properties_signify* schema, the translator parses perceivable property nodes and generates one record type for each set of perceivable properties having the same name (Fig. 7.4d), corresponding to $P_1 : \mathit{values} \setminus D_{2,\dots,n} \dots P_n : \mathit{values} \setminus D_{1,\dots,n-1}$. For a given BIGSIS-XML instantiation the translator generated up to 14 record types (i.e. $n \leq 14$), each corresponding to one of the 14 perceivable properties specifiable in the BIGSIS-XML grammar. Four such perceivable properties appear in Fig. 7.4a: *Symbol*, *Color*, *aPattern* and *Volume*. Therefore there are four record types represented in Fig. 7.4d: `Symbols_signify`, `Colors_signify`, `aPatterns_signify` and `Volumes_signify`.

Identifier-type pairs for each record type are generated by parsing *signifies* and *when-* attribute

values of perceivable property nodes. For all perceivable property nodes having the same name, the first identifier-type pair is generated from *when-* attribute values defines the property's set of end-user descriptions. Subsequent identifier-type pairs represent signified categories of function or meaning. Since all *signifies* attributes in perceivable property nodes of Fig. 7.4a are valued *"S1"* (referencing the *signified-meanings* node), one category identifier-type pair is represented in all four record types of Fig. 7.4d (`S1: S1_values`).

7.2.5.3 SAL Representation of *signifiers* Schema

To represent declarations of the *signifiers* schema, all *signifier-properties* nodes and *property-documentation* nodes are translated to record types corresponding to $C_{1,\ldots,n} : properties_signify_{1,\ldots,n}$ and $Doc_{1,\ldots,n} : properties_signify_{1,\ldots,n}$ respectively. A type for each node is constructed using its child perceivable property nodes. For all perceivable property nodes having the same name, exactly one corresponding identifier-type pair is added to each record type. For example, in Fig. 7.4a, *signifier-properties* has four sets of child nodes having the same name: *Symbol*, *Color*, *aPattern* and *Volume*. The translator produces a record type `signifiers_of_C1` at the top of Fig. 7.4e. This type corresponds to $C_1 : properties_signify_1$ of the *signifiers* schema. The accompanying documentation specified directly below it (`documentation_of_C1`) corresponds to $Doc_1 : properties_signify_1$. No SAL syntax is generated to represent the function declarations because one-to-one relationships between end-user descriptions and signified functions, meanings, or linked properties are already represented within perceivable property nodes: a node's *when-* attribute value and text content corresponds to $D_i \twoheadrightarrow S_i$.

The predicate part of the *signifiers* schema is represented within the SAL module in Fig. 7.4e. The translator generates local variables for each *signifier-properties* and *property-documentation* node, each corresponding to c_i and doc_i respectively within the predicate parts beginning $\forall\, c_i : C_i$, $\forall\, c_i : C_i;\ c_j : C_j$, and $\forall\, c_i : C_i;\ doc_i : Doc_i;\ doc_j : Doc_j$.

The translator does not generate SAL syntax representing the first predicate of the *signifiers* schema, which states that all perceivable properties are assigned an end-user description. A SAL model checker assigns end-user descriptions randomly at the outset of a model checking analysis,

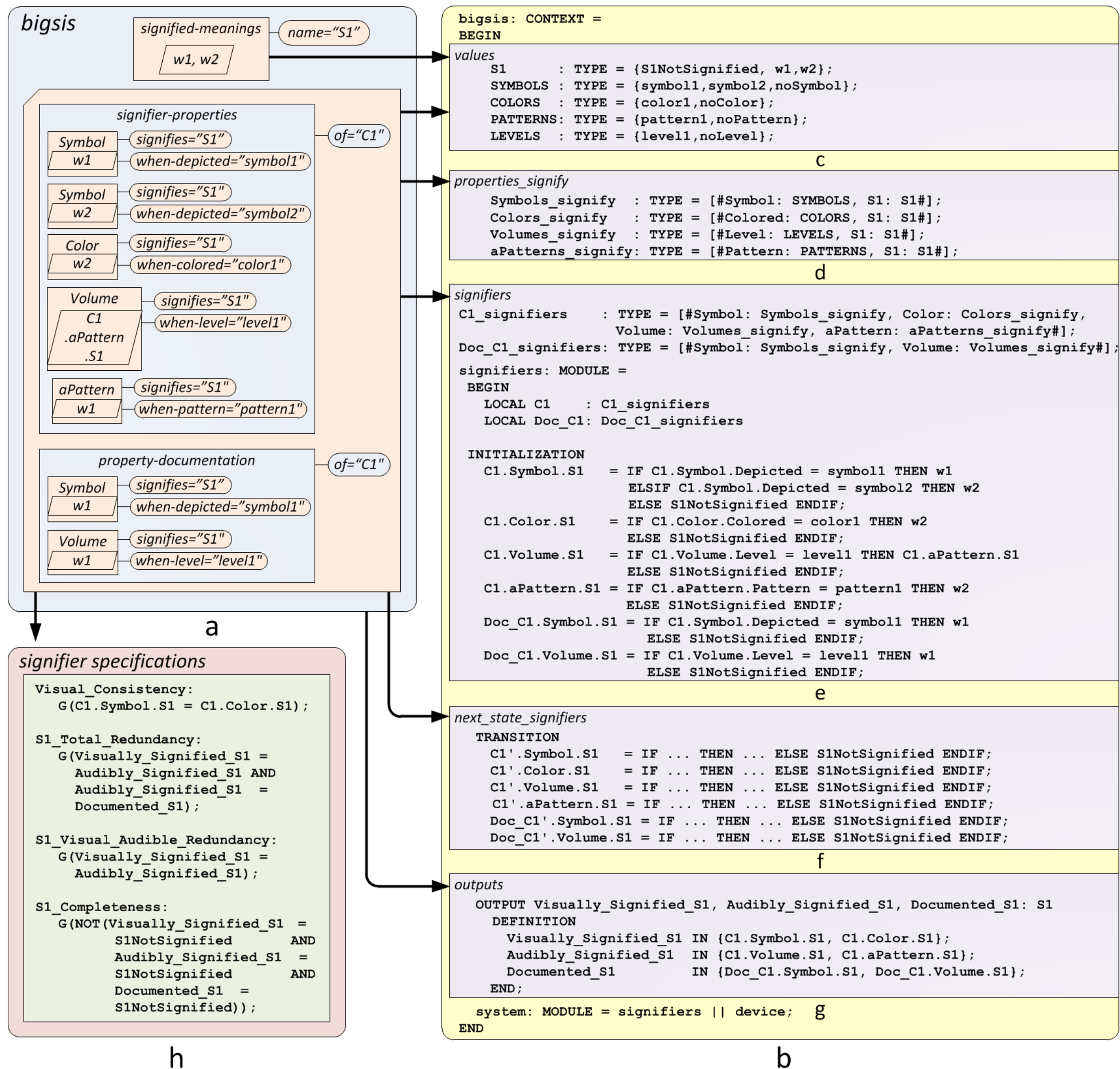


Figure 7.4: Visual representation of SAL code generated from a generic BIGSIS-XML instantiation. Arrows and bold-italic labels indicate which BIGSIS-XML nodes are parsed by the translator to represent SAL code. (a) Graphical representation of instantiated BIGSIS-XML. (b) Formal signifier model SAL code generated by the translator. (c–g) Portions of the translated SAL formal model corresponding to BIGSIS formalism schemas. (h) Four of the eight automatically generated LTL signifier specifications. Not shown: audible signifier consistency, documentation signifier consistency, visual and documentation partial signifier redundancy, audible and documentation partial signifier redundancy

but the analyst could optionally encode them manually.

SAL syntax representing the second and third predicates is generated under the `INITIALIZATION` heading in Fig. 7.4e. To represent the second predicate the translator parses all *signifier-properties* nodes and generates SAL conditional expressions. For each *signifier-properties* node, all perceivable

property child nodes having the same name and *signifies* attribute correspond to an instance of the predicate beginning $\forall\, c_i : C_i;\ c_j : C_j \bullet$ and ending **else** $c_i.P_i.S_i = w_0$. In the *signifiers* schema, one **if** $\cdots$ **then** $\cdots$ **else** ... expression is utilized to specify what is signified for all end-user descriptions (d_i). When these descriptions are instantiated in BIGSIS-XML, the corresponding SAL syntax satisfies the second *signifiers* schema predicate via a `IF ...THEN ...ELSIF ...THEN ...ELSE` conditional expression, where:

- The `IF ...THEN ...` part corresponds to the first perceivable property child of a *signifier-properties* node, and
- One `ELSIF ...THEN ...` part is generated for each perceivable property sibling node having the same node name (e.g. *Symbol*) and *signifies* attribute (e.g. *S1*)

Four such expressions are shown in Fig. 7.4e. Expressions representing the third predicate are generated in the same way for *property-documentation* nodes, and two are shown in Fig. 7.4e.

7.2.5.4 SAL Representation of *next_state_signifiers* Schema

The translator generates a set of SAL conditional expressions to represent the *next_state_signifiers* schema under the `TRANSITION` heading of Fig. 7.4f. Each next-state transition is a copy of the corresponding initialization statement with the addition of a "′" symbol. Since they are the same, `ELSIF` segments of conditional transition expressions are replaced with ... in Fig. 7.4f to conserve space.

The translator does not generate SAL syntax representing transitions to end-user descriptions. The analyst can encode these transitions manually using SAL syntax corresponding to BIGSIS formalism semantics. Alternatively, the analyst could encode model infrastructure considered separate from the BIGSIS formalism representing the system's control logic and/or human-system interaction. This technique is demonstrated in Section 7.3.

7.2.5.5 SAL Representation of *outputs* Schema

To represent the *outputs* schema, the translator parses perceivable property nodes and groups them into sets by channel (corresponding to the basic types [*vis*], [*aud*], [*hap*], and [*doc*]). The translator

generates SAL syntax for each set of perceivable property nodes in the same way, each of which represents one of four respective predicates. The translation protocol is described below with respect to the visual channel.

If the set of perceivable property nodes corresponding to the basic type $[vis]$ is empty (i.e. there are no nodes representing visually perceivable properties), SAL syntax representing the first predicate is not generated. Otherwise, one SAL output variable (corresponding to $visually_signified_i$: $signified_i!$) is generated for each category of function or meaning referenced within visually perceivable property node *signifies* attributes, and the output variable's name aids in identifying the channel and category (Fig. 7.4g, above `DEFINITION` heading). SAL selection statements (Fig. 7.4g, below `DEFINITION` heading) are generated to represent the predicate part beginning $\forall\, visually_signified_i$. Each SAL output variable listed on the left-hand side of `IN` corresponds to one $visually_signified_i$ variable of the outputs schema. All visually perceivable properties signifying the specified category of function or meaning (corresponding to $C.Vis.S_i$) are listed on the right-hand side. If there are no visually perceivable property nodes for the specified category of function or meaning in the BIGSIS-XML representation, SAL syntax is generated representing the predicate part $C.vis.S_i = \emptyset \Rightarrow visually_signified_i = w_0 \wedge w_0 \in S_i$. Such a case is not represented in Fig. 7.4e.

7.2.5.6 System Model Composition

The automatically generated system model employs the synchronous composition of device and signifier models (above `END` in Fig. 7.4). Such a composition ensures that end-user descriptions update correctly when the device changes states.

7.2.5.7 Automatic Generation of Signifier Specifications

Leveraging the semantics of LTL signifier specifications introduced in Section 7.2.3, the BIGSIS-XML-to-SAL translator also generates SAL theorems representing signifier consistency, redundancy, and completeness. Four such theorems are shown in Fig. 7.4h, automatically generated from the BIGSIS-XML instantiation represented in Fig. 7.4a.

As discussed in Section 7.2.3, a BIGSIS formalism representation having n categories of signified functions or meanings could have a maximum of four signifier consistency specifications (one for each

channel), n total signifier redundancy specification, $10n$ partial signifier redundancy specifications, and n signifier completeness specifications. These maximums are the same for SAL theorems generated by the translator. Specification generation rules are listed in outline form below with respect to BIGSIS-XML perceivable property nodes (Fig. 7.2e).

1. Signifier consistency:

   (a) Visual: two or more *Shape*, *Symbol*, *Label*, *Color*, *vPattern*, *oShape*, *oSymbol*, *oLabel* nodes have the same *signifies* attribute value and are direct children of *signifier-properties*. One specification is generated.

   (b) Audible: two or more *Pitch*, *Volume*, *aPattern* nodes have the same *signifies* attribute value and are direct children of *signifier-properties*. One specification is generated.

   (c) Haptic: two or more *Texture*, *Vibration*, or *hPattern* nodes have the same *signifies* attribute value and are direct children of *signifier-properties*. One specification is generated.

   (d) Documentation: two or more perceivable property nodes (i.e. any of those shown in Fig. 7.2e) have the same *signifies* attribute value and are direct children of *property-documentation*. One specification is generated.

2. Signifier redundancy:

   (a) Total: one or more visually perceivable property nodes (Fig. 7.2f), audibly perceivable property nodes (Fig. 7.2h), and/or haptically perceivable property nodes (Fig. 7.2i) have the same *signifies* attribute value. One specification is generated for each category of signified function or meaning.

   (b) Partial:

      i. Three-channel: All four channels are represented. One or more visually perceivable property nodes (Fig. 7.2f), audibly perceivable property nodes (Fig. 7.2h), and haptically perceivable property nodes (Fig. 7.2i) have the same *signifies* attribute value and are direct children of *signifier-properties* and *property-documentation*. One spec-

ification is generated for each category of function or meaning signified through all three channels.

i. Two-channel: At least three channels are represented. One or more visually perceivable property nodes (Fig. 7.2f), audibly perceivable property nodes (Fig. 7.2h), and haptically perceivable property nodes (Fig. 7.2i) have the same *signifies* attribute value and are direct children of *signifier-properties* or *property-documentation.* One specification is generated for each category of function or meaning signified through two channels.

3. Signifier completeness:

   (a) One or more visually perceivable property nodes (Fig. 7.2f), audibly perceivable property nodes (Fig. 7.2h), and haptically perceivable property nodes (Fig. 7.2i) have the same *signifies* attribute value and are direct children of *property-documentation.* One specification is generated for each category of function or meaning signified through the documentation channel.

7.2.6 Model Checking Technique

Specifications can be verified in SAL using either symbolic model checking (SAL-SMC) or bounded model checking (SAL-BMC) [68] (see Chapter 3, Section 3.2.3 for more information about these tools).

7.3 Case Study

The case study system is based on the left ventricular assist device (LVAD) described in Chapter 5. It includes a portable, battery-powered controller and printed documentation having labeled diagrams, tables, and instructional procedures. Using BIGSIS-XML, a subset of the controller's electrical/mechanical components, visual and audible properties presented on the device, and tables and diagrams within the patient handbook are represented. The end user is assumed to be an English-speaking adult having capabilities of perceiving visual and audible properties presented by

the controller, changing the device's settings, and understanding information within accompanying documentation.

To represent the system's control logic and human-system interaction, a device model considered separate from elements of the BIGSIS approach is encoded. The device model represents control logic controlling visual/audible properties and human-system interactions operating as inputs to guarded transitions representing next-states of end-user descriptions (corresponding to the *Deltasignifiers* schema). The system's interface is described in Section 7.3.1. The components, visual/audible properties, and documentation considered in the formal signifier model are described and encoded in the language of BIGSIS-XML in Section 7.3.2.

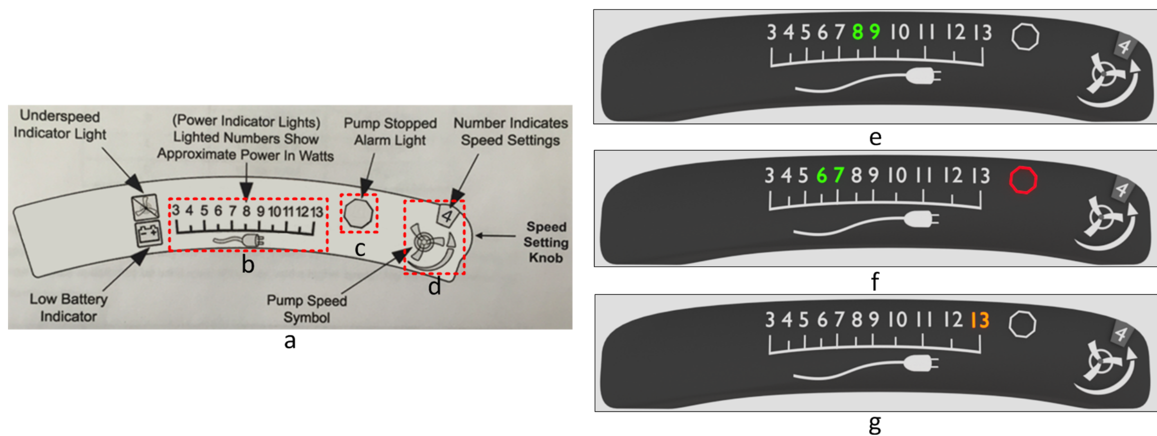


Figure 7.5: (a) Labeled diagram of the case study system's battery-powered controller appearing within accompanying documentation. Letters b–d and dashed red boxes are added to identify the three components considered in the case study. (b) Power indicator lights (numbered 3–12) and high power alarm (numbered 13). (c) Pump stopped alarm. (d) Speed setting knob. (e) Graphical rendering of the controller. The speed setting is four and the power indicator lights 8–9 are illuminated green. (f) High power alarm engaged: the number 13 illuminates amber and a loud, continuous alarm sounds. (g) Pump stopped alarm engaged: the octagon shaped light illuminates red and a loud, continuous alarm sounds

7.3.1 System Description

The system's controller has control logic for operating an implanted blood pump at one of five rotational speeds (numbered 1–5 from lowest to highest), and an end user could change the speed by rotating a speed setting knob on the side of the controller. The controller supplies as much power as necessary to the pump for maintaining a speed setting. The controller has control logic for illuminating any two adjacent power supply indicator lights numbered 3–12 (e.g. 8–9 in Fig. 7.5e)

and engaging one of four audible/visual alarms when a malfunction is detected. Two alarms are considered: the number 13 illuminates amber and a loud, continuous alarm sounds when the high power alarm is engaged (Fig. 7.5f); the octagon light illuminates red and a loud, continuous alarm sounds when the pump is rotating below a minimum threshold speed (Fig. 7.5g). Either audible alarm is emitted at the same pitch and volume.

The accompanying documentation describes what is signified by visual/audible properties of the components in Fig. 7.5. Text indicates that the high power alarm is the segment of power indicator lights labeled 13; and if the controller is supplying 13 or more watts to the pump, a loud, continuous alarm sounds while the number 13 illuminates amber. Text indicates that the pump stopped alarm light illuminates red and a loud, continuous alarm sounds if the pump is operating below 5,000 RPM. The underspeed alarm (Fig. 7.5a, top-left) engages when the pump is operating below its programmed speed and above 5,000 RRM, but it is not considered in the case study.

Table 7.2: Control knob settings, programmed speeds, and approximate power supplied by the controller discoverable within accompanying documentation

Setting	Programmed speed (RPM)	Power (watts)
1	8,000	3–4
2	9,000	4–5
3	10,000	5–6–7
4	11,000	7–8–9
5	12,000	8–9–10

A diagram (Fig. 7.5a) indicates that power indicator lights show power supplied to the pump in watts. Two tables (combined in Table 7.2) relate the speed setting knob's label to programmed rotational speeds and typical wattages supplied by the controller for each speed setting.

7.3.2 BIGSIS Model

A formal signifier model of the case study system represents three interface components described in the previous section (Section 7.3.1): the power indicator lights (including the high power alarm), the pump stopped alarm, and the speed setting knob. The model represents visual properties of color and label, audible properties of pattern and volume, text within accompanying documentation, and

tables within accompanying documentation (Table 7.2).

35 words/phrases are encoded to represent what is signified by the power indicators, pump stopped alarm, and speed setting knob; their colors, labels, audible patters, and volumes; and accompanying documentation describing what is signified. 12 of 35 words are encoded as text content within a *signified-meanings* node named *PumpSpeed*, six of which define relative pump speeds signified by the visual/audible properties of the pump stopped alarm and visual properties of the speed setting knob: *Stopped*, *Low*, *Lowest*, *Medium*, *High*, and *Highest*. The six other phrases define pump speed in RPM signified by accompanying documentation describing all three components: *BelowFiveThousandRPM*, *AboveFiveThousandRPM*, *EightThousandRPM*, *NineThousandRPM*, *TenThousandRPM*, *ElevenThousandRPM*, and *TwelveThousandRPM*.

23 of 35 phrases are encoded as text content within a *signified-meanings* node named *PowerSupplied*, ten of which define power supplied in units signified by visual/audible properties of the power indicators, including the high power alarm: *ThreeToFourUnits*, *FourToFiveUnits*, *FiveToSixUnits*, *SixToSevenUnits*, *SevenToEightUnits*, *EightToNineUnits*, *NineToTenUnits*, *TenToElevenUnits*, *ElevenToTwelveUnits*, and *ThirteenUnits*. 13 of 23 phrases define power supplied in watts signified by accompanying documentation describing power indicators, including the high power alarm, and the speed setting knob: *ThreeToFourWatts*, *FourToFiveWatts*, *FiveToSixWatts*, *SixToSevenWatts*, *SevenToEightWatts*, *EightToNineWatts*, *NineToTenWatts*, *TenToElevenWatts*, *ElevenToTwelveWatts*, *FiveToSevenWatts*, *SevenToNineWatts*, *EightToTenWatts*, and *ThirteenWattsOrGreater*.

Signifier-properties, *property-documentation*, *Color*, *Label*, and *aPattern* nodes of the BIGSIS-XML representation are described in outline form. The first outline describes what is signified through visual and audible channel properties of interface components described in Section 7.3.1, and the second outline describes what is signified through the documentation channel. To aid in associating outlined descriptions of what is signified with model variable names in the BIGSIS-XML representation, *signifier-properties* and *property-documentation* node *of* attributes are listed in italic text within parentheses.

The first outline describing what is signified by colors, labels, audible patterns, and volumes

presented on the device is shown below.

1. Power indicators (*signifier-properties of="PowerIndicators"*)

    (a) *Color*: amber signifies 13 power units supplied to the pump. Green signifies power units supplied by the pump, depending on what label is colored.

    (b) *Label*: any two adjacent numbers labeled 3–12 signify a range of power units supplied to the pump; for example, 4 and 5 illuminated in Fig. 7.5e signifies 4–5 units. The label 13 signifies 13 power units supplied to the pump.

    (c) *aPattern*: a continuous pattern signifies thirteen power units supplied to the pump.

    (d) *Volume*: a loud volume signifies thirteen power units supplied to the pump.

2. Pump stopped alarm (*signifier-properties of="PumpStoppedAlarm"*)

    (a) *Color*: red signifies that the pump is stopped. When no color is present, signified pump speed depends on the speed setting knob's label.

    (b) *aPattern*: a continuous pattern signifies that the pump is stopped.

    (c) *Volume*: a loud volume signifies the pump is stopped.

3. Speed setting knob (*signifier-properties of="SpeedSettingKnob"*)

    (a) *Label*: a number 1–5 signifies relative pump speed, lowest–highest

The second outline describing what is signified through the documentation channel is shown below.

1. Documentation of power indicators (*property-documentation of="PowerIndicators"*)

    (a) *Color*: amber signifies that 13 or more watts are supplied to the pump. Green signifies watts supplied by the pump, depending on what label is colored.

    (b) *Label*: any two adjacent numbers 3–12 signify watts supplied to the pump. The label 13 signifies 13 or more watts supplied to the pump. Any two adjacent numbers 3–8 signify pump speed (Table 7.2). Pump speed is not signified by the two adjacent numbers 8–9, since they appear within two rows of Table 7.2, corresponding to multiple pump speeds.

Two adjacent numbers 10–11 and 11–12 as well as the singular number 13 do not signify pump speed, since they do not appear in Table 7.2.

(c) *aPattern*: a continuous pattern signifies 13 or more watts supplied to the pump.

(d) *Volume*: a loud volume signifies 13 or more watts supplied to the pump.

2. Documentation of pump stopped alarm (*property-documentation of="PumpStoppedAlarm"*)

   (a) *Color*: red signifies that the pump speed is below 5,000 RPM.

   (b) *aPattern*: a continuous pattern signifies that the pump speed is below 5,000 RPM.

   (c) *Volume*: a loud volume signifies that the pump speed is below 5,000 RPM.

3. Documentation of speed setting knob (*property-documentation of="SpeedSettingKnob"*)

   (a) *Label*: a number 1–5 signifies pump speed in RPM, 8,000–12,000 in 1,000 RPM increments, and power supplied to the pump in watts (Table 7.2).

The BIGSIS-XML representation is encoded in 79 lines (Appendix G.1) and translated to 121 lines of SAL code using the automated tool described in Section 7.2.5.

7.3.3 Device Model

A device model represents human-system interaction and the LVAD's internal algorithms for the two considered alarms. This model controls changes to settings, alarms, and next-states of end-user descriptions (corresponding to the *next_state_signifiers* schema of the BIGSIS formalism). For human-system interaction, the device model represents a patient interacting with the speed setting knob to select a position labeled 1–5 or leave the position unchanged. For control logic, the device model represents states in which the pump stopped alarm is engaged, the high power alarm is engaged, or no alarm is engaged.

The initial *Alarm* value is *PumpStopped* and the initial *Action* value is *None*. For all subsequent values the alarm could transition randomly between *NoAlarm*, *PumpStopped*, and *HighPower* and the end user's interaction with the speed setting knob could transition randomly between *None*, *IncreaseSpeed*, and *DecreaseSpeed*.

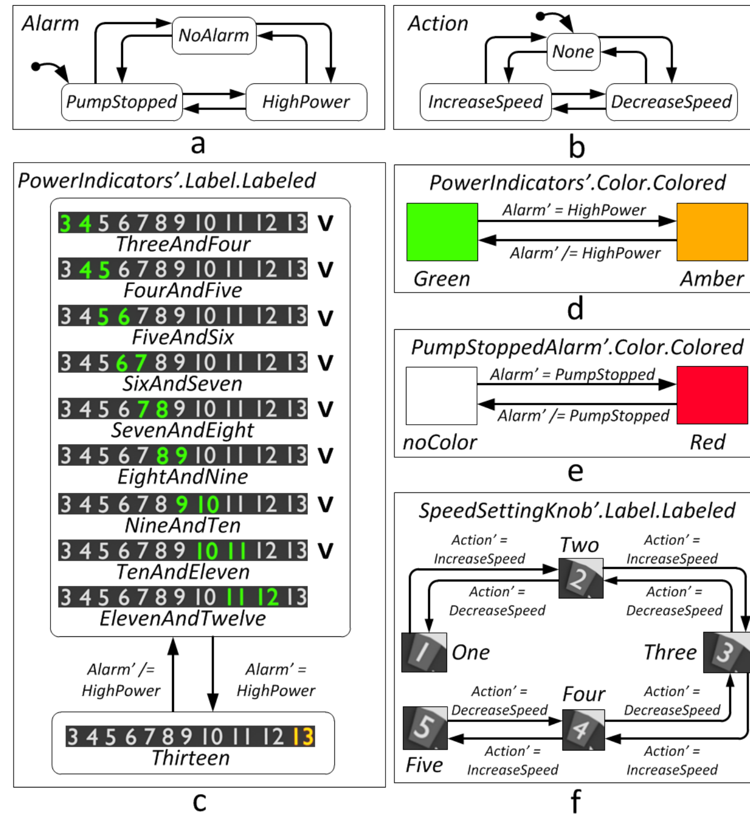


Figure 7.6: Graphical representation of the device model. Arrows connecting rounded-edge rectangles represent transitions. Curved arrows with filled circle represent initial states. (a) device model infrastructure representing the system's alarms. The variable *Alarm* is an output operating as an input to the *signifiers* module. (b) device model infrastructure representing human-system interaction. The variable *Action* is an output operating as an input to the SAL *signifiers* module. (c) Label of the power indicator lights. (d) Color of the power indicator lights. (e) Color of the pump stopped alarm light. (f) Label of the speed setting knob

The device model was 33 lines of SAL code (Appendix G.2.2). Outputs of the device model represented in Fig. 7.6a–b operate as inputs to the *signifiers* module, where guarded transitions control next-states of end-user descriptions. With this additional model infrastructure, the final BIGSIS-SAL model was 144 lines of SAL code (Appendix G.2.1).

7.3.4 Specifications

The translator generates 15 SAL theorems representing signifier specifications. For the purpose of demonstrating an application of the BIGSIS verification methodology, six are considered within model checking analyses. *Visual consistency* is specified to ensure that color of the pump stopped alarm and label of the speed setting knob are internally consistency with respect to pump speed,

while color and label of the power indicators are internally consistent with respect to power supplied. *Audible consistency* is specified to ensure that the audible pattern and volume of the pump stopped alarm are internally consistency with respect to pump speed, while audible pattern and volume of the high power alarm (incorporated within power indicators) are internally consistent with respect to power supplied. Two constrained, partial redundancy specifications are encoded to ensure that:

1. Visual and audible signifiers of power supplied have the same signified meaning when an alarm is engaged

2. Visual and documentation signifiers of pump speed have the same signified meaning when an alarm is engaged

States in which no alarms are engaged are not considered in the redundancy specifications because it is known that nothing will be signified through the audible channel in these states. Finally, two completeness specifications are encoded to ensure that:

1. A power supplied is always signified through at least one channel

2. A pump speed is always signified through at least one channel

The SAL syntax of each automatically generated specification is modified to compose signifier and device models synchronously, enabling them to transition together within model checking analyses. This composition ensures that outputs of the device model are in-sync with outputs of the signifier model.

7.3.4.1 Signifier Consistency

Two signifier consistency specifications are considered: *visual consistency* and *audible consistency*. The SAL syntax shown below for *visual consistency* reads, "colors and labels always signify the same pump speed and power supplied."

```
Visual_Consistency:
    G(PumpStoppedAlarm.Color.PumpSpeed  = SpeedSettingKnob.Label.PumpSpeed AND
    PowerIndicators.Color.PowerSupplied = PowerIndicators.Label.PowerSupplied);
```

The SAL syntax shown below for *audible consistency* reads, "audible patterns and volumes always signify the same pump speed and power supplied."

```
Audible_Consistency:
  G(PumpStoppedAlarm.aPattern.PumpSpeed  = PumpStoppedAlarm.Volume.PumpSpeed AND
  PowerIndicators.aPattern.PowerSupplied = PowerIndicators.Volume.PowerSupplied);
```

7.3.4.2 Signifier Redundancy

Two, two-channel partial signifier redundancy specifications are considered, both of which are modified as constrained specifications. The constraint ensures that model checking analyses only consider states in which an alarm is engaged. *PowerSupplied visual audible redundancy* reads, "When an alarm is engaged, visual and documented power supplied are always the same."

```
PowerSupplied_Visual_Audible_Redundancy:
    G(Alarm /= NoAlarm =>
        Visually_Signified_PowerSupplied = Audibly_Signified_PowerSupplied);
```

PowerSupplied visual documented redundancy reads, "When no alarm is engaged, visually signified and documented pump speeds are always the same."

```
PumpSpeed_Visual_Documented_Redundancy:
    G(Alarm = NoAlarm =>
        Visually_Signified_PumpSpeed = Documented_PumpSpeed);
```

7.3.4.3 Signifier Completeness

Two signifier completeness specifications are considered: *PowerSupplied completeness* and *PumpSpeed completeness*. *PowerSupplied completeness* reads, "it is never true that power supplied is not signified audibly, visually, or through accompanying documentation."

```
PowerSupplied_Completeness:
    G(NOT(Visually_Signified_PowerSupplied = PowerSuppliedNotSignified AND
        Audibly_Signified_PowerSupplied   = PowerSuppliedNotSignified AND
        Documented_PowerSupplied          = PowerSuppliedNotSignified));
```

PumpSpeed completeness reads, "it is never true that pump speed is not signified audibly, visually, or through accompanying documentation."

```
PumpSpeed_Completeness:
    G(NOT(Visually_Signified_PumpSpeed = PumpSpeedNotSignified AND
        Audibly_Signified_PumpSpeed   = PumpSpeedNotSignified AND
        Documented_PumpSpeed          = PumpSpeedNotSignified));
```

7.3.5 Verification

Specifications were verified using SAL's symbolic model checker (SAL-SMC) [68]. Results, number of states visited, and execution times are shown in Table 7.3. Signifiers could be considered safe with respect to audible signifier consistency, power supplied signifier completeness, and pump speed signifier completeness, but unsafe with respect to visual signifier consistency and partial signifier redundancy.

Table 7.3: Case study model checking results

Specification name	Result	States visited	Execution time (s)
Visual consistency	*counterexample*	$5.5465277668511 \times 10^{23}$	1.31
Audible consistency	*proved*	$3.1448812438046 \times 10^{26}$	3.90
PowerSupplied visual audible redundancy	*counterexample*	$5.5465277668511 \times 10^{23}$	1.20
PumpSpeed visual documented redundancy	*counterexample*	$7.0256018380114 \times 10^{23}$	1.15
PowerSupplied completeness	*proved*	$3.1448812438046 \times 10^{26}$	3.96
PumpSpeed completeness	*proved*	$3.1448812438046 \times 10^{26}$	3.92

The counterexample to *visual consistency* represents the initial state in which the pump stopped alarm colored *Red* signifies a pump speed of *Stopped*, while the speed setting knob labeled *Three* signifies a pump speed of *Medium* (Fig. 7.7). This state represents a potential understandability failure, as the speed setting knob label *Three* and the pump stopped alarm color *Red* signify conflicting pump speeds concurrently.

Figure 7.7: Visualization of case study counterexample to *visual consistency*. Variables and values are listed in italic text within rectangles. Arrows point from variables to interface components. Conflicting meanings are listed in boldface white text within black rectangles

The counterexample to *PowerSupplied visual audible redundancy* represents the initial state in which the visually signified power supplied is *SixToSevenUnits* and the audibly signified power sup-

plied is *ThirteenUnits* (Fig. 7.8). Power indicators labeled *SixAndSeven* signify *SixToSevenUnits* while the pump stopped alarm volume level *Loud* and audible pattern *Continuous* signify *ThirteenUnits*. This state reflects a potential understandability failure, as there are conflicting meanings regarding power supplied signified concurrently through audible and visual channels.

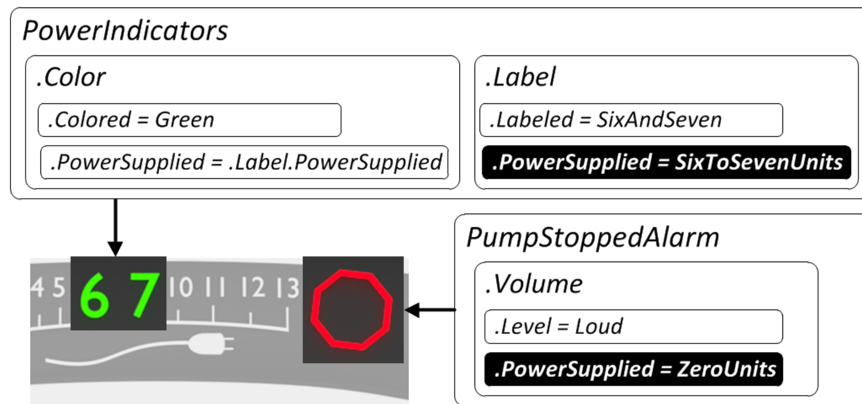
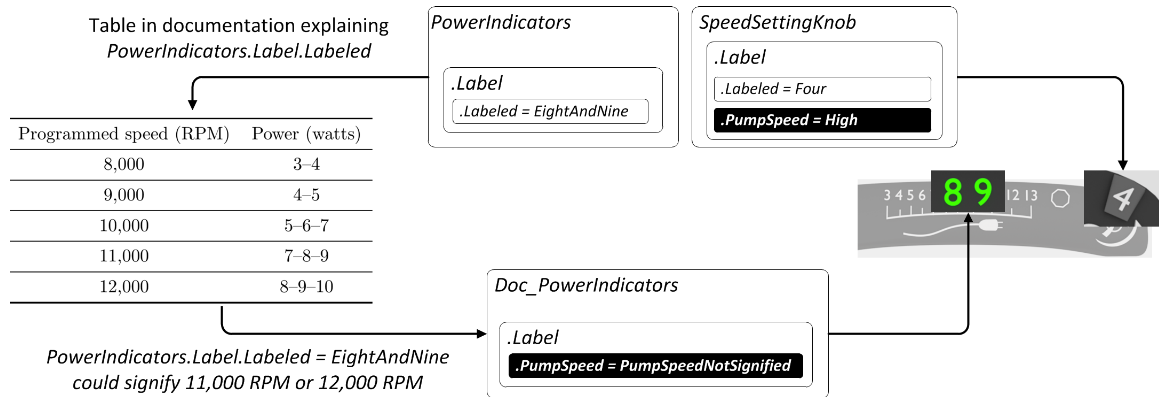


Figure 7.8: Visualization of case study counterexample to *PowerSupplied visual audible redundancy*. Variables and values are listed in italic text within rectangles. Arrows point from variables to interface components. Conflicting meanings are listed in boldface white text within black rectangles

The counterexample to *PumpSpeed visual documented redundancy* represents a one-step trace through the model in which the visually signified pump speed is *High*, but the documented pump speed is *SpeedNotSignified* (Fig. 7.9). The speed setting knob labeled *Four* signifies a pump speed of *High* through the visual channel, while pump speed is not signified through the documentation channel. This is due to the tables in the patient handbook indicating that when the power indicators are labeled 8–9, the pump speed could be either 11,000 or 12,000 RPM. Because such a one-to-many relationship is not specifiable in the BIGSIS formalism, pump speed is not signified through the documentation channel when the end-user description of label for the power indicators is *EightAndNine*. This state reflects a potential understandability failure: conflicting pump speeds are signified concurrently through visual and documentation channels.

7.4 Scalability

In the case study model, a subset of next-state end-user descriptions were enabled via the exchange of input/output variables with an auxiliary *device* model. However, it could be possible for a human-



Programmed speed (RPM)	Power (watts)
8,000	3–4
9,000	4–5
10,000	5–6–7
11,000	7–8–9
12,000	8–9–10

Figure 7.9: Visualization of case study counterexample to *PumpSpeed visual documented redundancy*. Variables and values are listed in italic text within rectangles. Arrows point from variables to interface components. Conflicting meanings are listed in boldface white text within black rectangles

system interface to support all end-user descriptions, and formal verification of signifier models should be scalable for such an interface. Thus, scalability of the BIGSIS approach was evaluated to assess the effects on model size and symbolic model checking verification times with respect to:

- One-to-one relations
- Guarded transitions enabling all end-user descriptions

A base model and five increasingly larger models were encoded in BIGSIS-XML, translated to SAL, augmented with initial- and next-state end-user descriptions, and verified using symbolic model checking. BIGSIS-XML nodes of the base model are described in outline form below:

- two *signified-meanings* nodes having one word each as text content
- one *signifier-properties* node having:
  - one audibly perceivable property child node having one end-user description and one signified meaning
  - one visually perceivable property child node having one end-user description and one signified meaning

This model encompasses two one-to-one relations. The base model was enlarged by adding one additional comma-separated word to each *signified meanings* node, duplicating the perceivable property

nodes, and modifying the *when*- attribute/text content of duplicated nodes such that two new one-to-one relations were introduced. This technique does not modify the number of *signified meaning* nodes or channels represented in perceivable property nodes, which constrains the scalability evaluation to guarded transitions and one-to-one relations. This process was repeated four times to generate six BIGSIS-XML representations. Each BIGSIS-XML representation was translated to SAL; one set of initial end-user descriptions was assigned (i.e. one for each property of each components); and two guarded transitions were encoded for each one-to-one relation such that all descriptions could be assigned in subsequent states:

1. The 1st (base) model has two one-to-one relations and four guarded transitions
2. The 2nd model has four one-to-one relations and eight guarded transitions
3. The 3rd model has eight one-to-one relations and 16 guarded transitions
4. The 4th model has 16 one-to-one relations and 32 guarded transitions
5. The 5th model has 32 one-to-one relations and 64 guarded transitions
6. The 6th model has 64 one-to-one relations and 128 guarded transitions

An LTL specification (encoded generally in (7.8)) was developed for each model to ensure that all reachable states are enumerated verification. The specification, which is always true, asserts that the one component's perceivable property always signifies one meaning from the set of possible meanings represented in the model.

$$\mathsf{G}(C_i.P_i.S_i = w_0 \vee \ldots \vee C_i.P_i.S_i = w_n) \tag{7.8}$$

Specifications were verified using SAL-SMC [68]. The number of states visited and verification times for each model are reported in Table 7.4. Doubling the number of one-to-one relations and guarded transitions increased model size and verification time exponentially. On the target work-

station, model checking succeeded in the first five models, but failed on the 6th model having 64 one-to-one relations and 128 guarded transitions.

Table 7.4: Results of scalability evaluation. "—" indicates that model checking failed

One-to-one relations	Guarded transitions	States visited	Verification time (s)
2	4	4	0.06
4	8	49	0.08
8	16	9.37890625×10^{8}	0.35
16	32	$3.595305184641 \times 10^{12}$	7.01
32	64	$3.9402006196395 \times 10^{115}$	328,195.36
64	128	—	—

7.5 Discussion

This chapter introduced a novel, formal methods based approach for analyzing interface signifiers, including:

1. The BIGSIS formalism and modeling technique
2. LTL signifier specifications based on safety-critical system usability standards
3. An encoding tool that facilitates the development of formal models and LTL signifier specifications

The approach was demonstrated in a medical device case study, and scalability of the approach was evaluated using symbolic model checking. Methodological considerations and directions of future work are discussed in the following sections.

7.5.1 The BIGSIS Formalism and Modeling Technique

The BIGSIS formalism is the first attempt of representing signifiers within a formal model. In determining what is signified the analyst is meant to consider end-user knowledge of the system, cultural context, and perceptual capabilities. By representing what is signified as outputs of the human-system interface, the BIGSIS formalism and modeling technique force the analyst to reason about interactions between the user and device. Similar to how the process of encoding formal models

has proven useful in other applications [101], the process of instantiating the BIGSIS formalism could be useful for informing the design of visual, audible, and haptic properties of interface components that support understandability for a specified end user.

By representing information within a system's accompanying documentation as a signifier channel, the BIGSIS formalism provides a new way of incorporating documentation as part of the interface. Other formal methods-based frameworks do not represent documentation in this way. For example, the EOFM task analytic formalism [10] was useful for analyzing how device descriptions could influence end-user task behavior in Chapter 5; however, device descriptions were only applicable within the context of a procedure. Additionally, interactions among user manual text, perceivable properties of the device, and end-user interpretation were difficult to model. The BIGSIS formalism provides these modeling capabilities.

The relation and explanation functions of the *signifiers* schema enable the analyst to specify what is signified by individual properties of interface components. Currently, these functions are limited to specifying what is signified by one component, property, and end-user description. Researchers in semiotics have demonstrated that signifiers could operate as hierarchical systems in which many components and properties interact (see for example [213]). There is no way to represent these phenomena using the BIGSIS formalism. If multiple properties and components are necessary for a function or meaning to be signified, the analyst must abstract their interaction by specifying linked properties. Additionally, these interactions are specifiable in a heterarchical way. Abstracting interactions between signifiers by linking them proved useful in the case study; however, it could be beneficial to refine the BIGSIS formalism in future work such that a broader range of interactions among components and properties can be modeled.

7.5.2 Signifier Specifications

The signifier specifications introduced in this research integrate the formal semantics of LTL with the theory of signifiers from HCI [37]. Generalizable specifications assert *signifier consistency* and *signifier redundancy*, which could be useful for verifying two aspects of interface understandability early in the design cycle. The *signifier completeness* specification could be useful for ensuring that

if perceivable properties on the device are insufficient, accompanying documentation provides the missing information. For system models that utilize separate representations of displays, controls, or human-system interaction, constrained specifications enable the analyst to verify signifier specifications for particular states of interest, such as those in which a certain alarm is engaged. This encoding technique was inspired by the approach discussed in [46].

While these specification proved useful in the case study, they should be validated empirically in future work to determine if model checking counterexamples represent real usability problems for human study participants. Additionally, the specifications do not capture all five safety-critical system usability measures identified in Chapter 1. For example, the understandability- and completeness-related specifications do not consider whether what is signified is correct with respect to the system's operational state (i.e. whether signifiers are accurate). One way of specifying signifier accuracy is addressed in Chapter 9. The applicability of other safety-critical system usability measures with respect to LTL signifier specifications should be explored in future work.

7.5.3 Encoding Tool: BIGSIS-XML and Translator

BIGSIS-XML is the first formal grammar developed to facilitate formal signifier modeling. Leveraging theories of perception from psychology and HCI, a constrained set of keywords identifies different kinds of interface component properties that could be perceived and identified in different ways. While the tool was developed to facilitate formal model development, it could be useful on its own for evoking design considerations during the encoding process. For example, while encoding the BIGSIS-XML representation of the medical device interface, it was observed that the numbers 8–9 illuminated green on the power indicators could signify two pump speeds through the documentation channel, 11,000 and 12,000 RPM (Table 7.2). Therefore, accompanying documentation may not support the end user in identifying a pump speed if the numbers 8–9 are illuminated. Uncovering this potential problem was possible because the formal semantics of BIGSIS-XML require the analyst to encode one-to-one relationships between end-user descriptions and signified functions or meanings. This concern is revisited in Section 7.5.4.

The constrained set of visual, audible, and haptic properties that are specifiable in BIGSIS-XML

proved useful in the case study (discussed further in Section 7.5.4), but it could be beneficial to explore a broader range of properties in future work. For example, keywords for specifying the spatial position of components and properties with respect to each other could improve applicability of BIGSIS-XML to a broader range of systems. Another area of interest involves properties that perceivable through multiple channels, such as length, width, and depth dimensions that are both kinesthetic and visual. These properties are critical for understanding what motor actions the end user could execute to manipulate interface components, such as whether a button appears pushable based on its depth and diameter. Incorporating these properties within the BIGSIS-XML grammar could enable integrated analyses of signifiers and, leveraging the methodologies from Chapter 6, affordances. For example, it could be useful for a BIGSIS-XML representation to specify whether a button's dimensions signify "pushable," while a CAVEMEN-XML representation specifies what human-environment system constraints enable the "pushable" affordance. Models could be translated and verified to ensure that pushability is always signified and afforded. Extensions to BIGSIS-XML supporting such a technique should be explored in future work.

To facilitate model checking analyses of signifiers, BIGSIS-SAL models and LTL signifier safety specifications are automatically generated by the automated translation tool. One reason the translator is advantageous is because it makes model checking analyses possible without the need for SAL syntax. However, knowledge of XML is still needed, and additional SAL infrastructure needs to be encoded manually if the analyst wants to model end user-device interaction. Formal methods researchers have developed GUI-based tools for generating formal model code from visual representations (see for example [48]), and it could be beneficial to develop such a tool for BIGSIS in future work.

7.5.4 Medical Device Case Study

Case study results indicate that the BIGSIS approach shows promise for enabling formal verification of signifiers for an interface having colors, labels, patterns, volumes, and accompanying documentation explaining what is signified. Modeling checking counterexamples proved useful for uncovering potential consistency and redundancy problems, while the process of instantiating a BIGSIS-XML

model aided in identifying what model checking analyses should be conducted. For example, the *PowerSupplied visual documented redundancy* specification was selected to revisit a potential understandability problem uncovered during the BIGSIS-XML encoding process: power indicator lights labeled 8–9, corresponding to one end-user description of label, relate to multiple pump speeds in Table 7.2; thus, in the BIGSIS-XML representation, a pump speed was not signified through the documentation channel. The *PowerSupplied visual documented redundancy* specification was encoded to investigate this concern. Its counterexample revealed that documentation could introduce a potential understandability because it provides information that conflicts with the pump speed signified by the speed setting knob label. Therefore, the sequential approach of flagging the potential problem during the BIGSIS-XML encoding process, revisiting it in a safety specification, and analyzing it via model checking proved useful.

While encoding guarded transitions for end-user descriptions, it was observed that the power indicator label description of "*ElevenAndTwelve*" is never assigned (i.e., numbers 11–12 on the controller are never illuminated green). In Table 7.2, the numbers 11–12 are unused and the numbers 8–9 are used twice to provide two signified pump speeds, which is equivalent to no signified pump speed. The process of encoding a separate *device* model as part of the BIGSIS approach helped identify this potential problem.

7.5.5 Scalability

Scalability of BIGSIS-SAL models was evaluated using symbolic model checking in models having up to 128 guarded description transitions and 64 one-to-one relations between end-user descriptions and signified meanings. Results indicate that scalability is limited with respect to SAL-SMC verification time in models having up to 32 one-to-one relations and 64 guarded transitions, while a 64 GB workstation cannot support SAL-SMC in models having as many as 64 one-to-one relations and 128 guarded transitions. In future work, it could be beneficial to explore scalability of the BIGSIS approach using other model checking tools, such as bounded model checking.

Chapter 8: A Formal Approach to Controlled Actuators: Modeling of Continuous Device Dynamics Derived from Spreadsheet Data

The model checking methodologies discussed in Chapters 4–7 have dealt with discrete models of system and human operator behaviors, building toward the integrated framework outlined in Chapter 1. To complete this framework, an approach for incorporating continuous device behavior within model checking analyses is needed.

Several model checking tools and techniques have proven useful for representing continuous device behavior in HFE [46, 76, 77, 214, 79]. In such applications, formal models utilize abstract representations of differential equations [119], and outputs are approximate solutions resembling those captured by numerical and analytical methods. Solutions could be utilized to represent continuous device behaviors considered possible in the actual system (see for example [215]); however, there are often more behaviors represented in an abstract model than are actually possible in the underlying system.

In other engineering domains, data from computational simulations and tests (e.g. benchtop medical device tests performed in a laboratory) are often collected throughout the design cycle to evaluate critical performance characteristics [216]. Computational models leveraging differential equations are often utilized, and the solutions they produce are precise calculations of actual device behavior. Examples include pressure and flow measurements for an implantable blood pump [87] and therapy rendered in a patient-specific physiology [217].

In regard to safety-critical systems, an advantage of simulation and laboratory testing is precision with respect to critical performance; however, it could be difficult leverage these data within analyses that consider the interface and human operator. As discussed in Chapter 1, advantages of formal methods are inclusiveness with respect to human-system interface models; however, differential equation abstractions of continuous device dynamics are imprecise. Considering complementary characteristics of simulation/testing and formal methods, it could be beneficial for analysts to com-

bine them.

For such a combined approach to be possible, data collected in simulations and tests must be represented formally using model checking syntax. Such data are often stored in tabulated spreadsheet formats such as .xls (Microsoft Excel) or .csv (**c**omma-**s**eparated **v**alues) having rows and columns of cells. A common way of organizing tabulated data is by representing one group of similar measurements (e.g. blood glucose, heart rate) in a column with individual numbers in cells. If there are multiple kinds of measurements, they are typically organized in a row of cells. It is also possible to tabulate data using rows, columns, and sets of columns called slices (Fig. 8.1a). Analysts recording data in Microsoft Excel, for example, may use tabs (considered slices) to store data for different experimental conditions such as device settings. For example, the graphical representation of medical device testing data in Fig. 8.1b has two kinds of measurements (flow and wattage) in five slices representing different settings (rotational speed setting).

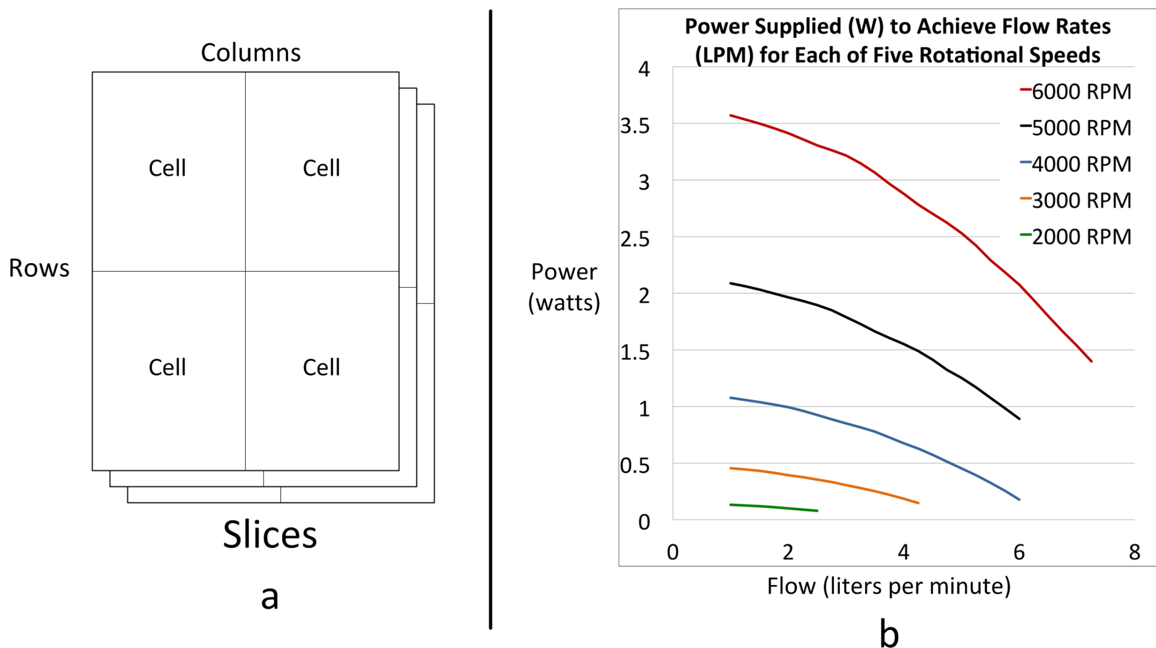


Figure 8.1: (a) Graphical representation of tabulated data having rows, columns, and slices. (b) Graphical representation of blood pump testing data having two kinds of measurements (flow rate on the x-axis and power on the y-axis) and five slices (rotational speed represented by each line)

Considering common characteristics of testing data, a tool supporting model checking analyses that incorporate them should be capable of parsing spreadsheet files containing numeric simula-

tion/testing data. It should generate model checking syntax representations in a way that separates cells within one or more rows, columns and slices. It could also be beneficial for these representations to support model checking techniques that have proven useful, such as relational abstraction [46].

This chapter describes such a tool. Utilizing a custom MATLAB function, a spreadsheet file containing data from prior simulations/tests is translated to SAL syntax representing its contents. This SAL syntax leverages the relational abstraction technique described in Chapter 3, Section 3.2.2 by generating a *constraints* model. The analyst can encode a separate SAL model representing continuous device elements (i.e. a *plant* model), and these two can be composed in a continuous device model. Instead of matching abstract differential equation behaviors, continuous device model outputs match simulation or testing data represented in the spreadsheet file.

The MATLAB-based translation function, the format of spreadsheet data required by the function, and SAL syntax of translated representations are discussed in the following sections. The approach is demonstrated by augmenting the relational abstraction technique described in [46] with a SAL representation of testing data for a medical device under development. A scalability evaluation is conducted using exemplar data generated in MATLAB, and a discussion of methodological considerations and future work follows.

8.1 Modeling Methodology: Representing Testing Data Formally

The MATLAB-based translation tool is a function having three inputs:

1. A .xls, .xlsx, or .csv spreadsheet containing testing data (e.g. *C:\tests\test1.xlsx*)
2. An optional integer specifying the number of columns per slice (explained later in this section)
3. A name for the SAL context being generated (e.g. *test1*)

If the spreadsheet file containing test data has slices representing different testing conditions, such as device settings, data for each slice could be organized in multiple spreadsheet tabs (one for each slice) within a single .xls or .xlsx file. Utilizing such an organization, each spreadsheet must have the same number of columns, and each cell must contain a number. If a cell does not contain a number, the function inserts a zero. If spreadsheets have different numbers of rows, the function

inserts zeros in empty rows. If such an organization is utilized within a .xls or .xlsx file, the second function input is not required.

If the spreadsheet containing test data has slices and is a .csv file, the analyst must organize slices in sets of parallel columns, since the .csv format does not support multi-spreadsheet files. If desired, the analyst could optionally organize slices within a single .xls or .xlsx file spreadsheet in this way. If such an organization is utilized, the second function input must be an integer specifying the number of columns per slice.

The filename is specified in the second input (or third if the parallel column organization is utilized), and the analyst should choose a filename that aids in identifying the testing data being translated to SAL. The tool automatically appends the extension *.sal* to this input and saves the translated SAL model in the current MATLAB directory.

The translated SAL model has Boolean functions representing slices, where inputs to the functions represent data in each slice. One row of cells within a slice is translated to a set of Boolean function inputs creating an output value of *true*. If a Boolean function's inputs do not match data within at least one row, its value is *false*. The automatically generated *constraints* model sets a Boolean variable named *fitsData* to *true* if all Boolean functions representing slices produce output values of *true*. Otherwise, *fitsData* is set to *false*. This translation protocol and the formal semantics of generated SAL models are explained using an instantiated example in the next section.

8.1.0.1 Formal Semantics of Translated SAL Models

For a spreadsheet file having s slices and c columns per slice (where s and c are integers), a translated SAL model represents slices within s Boolean-valued functions. Each function takes a single input named `inVars` (called a tuple) of length c (i.e. a c-tuple). Each element of the c-tuple `inVars` has the SAL type `REAL`, which corresponds to any real number, positive or negative, having up to ten decimals places. The general encoding of such Boolean valued functions for a spreadsheet file having s slices is shown below.

```
slice_1(inVars:  [REAL,..., REAL]): BOOLEAN =
                         ...
slice_s(inVars:  [REAL,..., REAL]): BOOLEAN =
```

For each function representing a slice (referred to as a slice Boolean), each row is represented by a value of `inVars`. This value is a set of comma-separated numbers corresponding to all cells in a row. Each set of comma-separated values is separated by the SAL operator `OR`, specifying that the slice Boolean is valued *true* if `inVars` is equal to a row of values from the spreadsheet. A general encoding is shown below for a slice Boolean representing the first slice of testing data having r rows and c columns. Text is utilized instead of numbers (`cell_1,..., cell_c`, where `cell_c` is contained within a row's last column) to represent numeric values contained within cells. Parenthesized text is added to aid in identifying SAL syntax corresponding to spreadsheet rows.

```
slice_1(inVars:  [REAL,..., REAL]): BOOLEAN =
     inVars = (cell_1, ...,  cell_c) OR (row 1)
                        ...             OR (rows between 1 and r)
     inVars = (cell_1, ...,  cell_c);   (row r)
```

The automatically generated *constraints* model is a SAL module whose name is the third MATLAB function input (the translated SAL context name) with `_Constraints` added at the end. The module has one local variable for each column in a slice (`var_1,..., var_c` for a data having c columns and one or more slices) and one output variable named `fitsData` by default. One guarded initialization command and one guarded transition command (Chapter 3, Section 3.2.1.1) are generated for each slice.

Each initialization guard asserts that local variable values are inputs to each slice Boolean, and if those inputs make a slice Boolean *true*, the initial value of `fitsData` is true. Otherwise, `fitsData` is *false* (asserted in a final `[]ELSE` guard). This initialization command syntax is encoded generally below.

```
INITIALIZATION [
  slice_1(var_1,..., var_c)   -->
    fitsData = true;
           ...
  []slice_c(var_1,..., var_c) -->
    fitsData = true;
  []ELSE -->
    fitsData = false;
];
```

Generated guarded transition commands represent the opposite conditions for next-states: if the next-states of local variable values make a slice Boolean *false* (denoted by `not` in the general

encoding below), the next-state of `fitsData` is *false*. Otherwise, the value of `fitsData` remains unchanged (asserted in the `[]ELSE` guard triggering no next-state assignment).

```
TRANSITION [
  not(slice_1(var_1',..., var_c'))   -->
    fitsData' = false;
           ...
  []not(slice_c(var_1',..., var_c')) -->
    fitsData' = false;
  []ELSE -->
];
```

To demonstrate the syntax of a complete SAL context, an example model is stantiated based on hypothetical spreadsheet data having two slices, two columns, two rows, and eight cells (Table 8.1). For such a dataset, inputs to the MATLAB-based translation function are:

1. *example.csv*, specifying the filename of hypothetical data

2. *2*, specifying that *example.csv* has two columns per slice organized in parallel

3. *example*, specifying the name of a generated SAL context (filename *example.sal*)

The generated SAL syntax is shown below Table 8.1. Parenthesized text is added to aid in identifying what SAL syntax corresponds to the data.

Table 8.1: Tabulated representation of hypothetical testing data in a file named *example.csv*

slice 1		slice 2	
1.049	0.554	1.296	0.347
0.998	2.717	2.101	1.816

```
example: CONTEXT =
BEGIN
  slice_1(inVars: [REAL, REAL]): BOOLEAN =
    inVars = (1.049, 0.554) OR  (First row in slice-1 of Table 8.1)
    inVars = (0.998, 2.717);  (Second row in slice-1 of Table 8.1)

  slice_2(inVars: [REAL, REAL]): BOOLEAN =
    inVars = (1.296, 0.347) OR  (First row in slice-2 of Table 8.1)
    inVars = (2.101, 1.816); (Second row in slice-2 of Table 8.1)

  example_Constraints: MODULE =
  BEGIN
    LOCAL var_1: REAL
    LOCAL var_2: REAL
    OUTPUT fitsData: BOOLEAN
    INITIALIZATION [
      slice_1(var_1, var_2)  -->
        fitsData = true;
      []slice_2(var_1, var_2) -->
        fitsData = true;
      []ELSE -->
        fitsData = false;
    ];
    TRANSITION [
      not(slice_1(var_1', var_2'))   -->
        fitsData' = false;
      []not(slice_c(var_1', var_2')) -->
        fitsData' = false;
      []ELSE -->
    ];
  END;
END
```

8.1.1 Verification Methodology

To conduct model checking analyses, the analyst could encode a separate *plant* model representing continuous behaviors of the system that produced spreadsheet data. Such a model could employ any of the continuous formal modeling techniques discussed in Chapter 1 or a different representation that does not employ differential equations. The only requirements of a *plant* model are that its outputs are real numbers, and at least one output is represented in spreadsheet data.

A graphical representation of the model architecture employed within model checking analyses is shown in Fig. 8.2. The *plant* and *constraints* models are synchronously composed so they transition at the same time, creating a continuous device model that synchronizes local variables of *constraints* and output variables of *plant* in-sync (Fig. 8.2a). Outputs of the *plant* model (represented generally as `Output_1,..., Output_n`) must be inputs to the *constraints* model, and one or

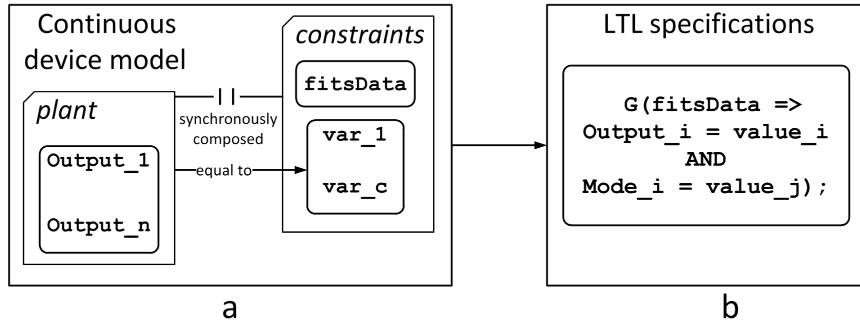


Figure 8.2: A graphical representation of continuous model architecture employed within model checking analyses. (a) A continuous device model having synchronously composed *plant* and *constraints* models. (b) One or more LTL specifications encoded using continuous device model outputs, where `Output_i = value` is a generic *plant* model state that should always hold

more automatically generated local variables are set equal to these inputs.

Continuous device model outputs are utilized within LTL specifications (Fig. 8.2b). The *constraints* model output variable `fitsData` is utilized to force the model checker to only consider *plant* model outputs matching simulation data. This LTL specification encoding technique is based on the relational abstraction approach described in Chapter 3, Section 3.2.2.

8.2 Case Study

To demonstrate an application of the approach, a case study was developed using computational fluid dynamics simulation data from a medical device under development. The device, testing data, a SAL context translated from testing data, and additional SAL model infrastructure supporting model checking analyses are discussed in this section. A *plant* model is encoded as an input-output function, and a *constraints* model is generated using the MATLAB-based tool. These models are composed within continuous device model, and SAL's infinite bounded model checker (SAL-INF-BMC) [68] is utilized to verify that output values match simulation data.

8.2.1 The Device

The case study device under development is intended for pediatric patients lacking a functional right ventricle, which normally pumps oxygen-poor blood from bodily circulation to the lungs. Such patients typically undergo a series of surgical procedures that reconfigure circulatory anatomy in a

way that bypasses the right ventricle and goes directly to the lungs, resulting in a condition called Fontan physiology [218].

A mechanical circulatory support device is often necessary to render therapy in patients with Fontan physiology as a bridge to recovery or heart transplantation [219]. Engineered to provide such therapy, the case study device is an intravenous blood pump intended for implantation in a major vein carrying oxygen-poor blood from the lower half of the body directly to the lungs. It augments blood pressure and flow using a rotating impeller, a stationary protective cage, and a stationary diffuser (Fig. 8.3).

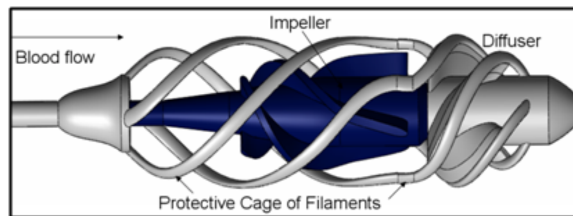


Figure 8.3: Graphical rendering of the case study device. Reprinted from [15]

8.2.2 Testing Data

One critical design target for the case study device is electrical power in watts required by the pump, which depends on many interacting factors.Two are considered in this case study: impeller speed in rotations per minute (RPM) and blood flow augmentation in liters per minute (LPM).

Power usage data were collected as a dependent measure for independent measures of flow rate and impeller speed using computational fluid dynamics simulation (ANSYS CFX) [220]. Five rotational speeds ranged from 2,000–6,000 RPM in 1,000 RPM increments and 26 flow rates ranged from 1–7.25 LPM in 0.25 LPM increments. Data were organized in a spreadsheet (represented graphically in Fig. 8.1b) having five, two-column slices, where each slice represents data for a different rotational speed. Flow rates are placed in the first column of each slice and power usage measurements are placed in the second column.

Rotational speeds and flow rates tested were 2,000 RPM for 1–2.5 LPM, 3,000 RPM for 1–4.25 LPM, 4,000 for 1–6 LPM, 5,000 RPM for 1–6 LPM, and 6,000 RPM for 1–7.25 LPM. These data (Table 8.2) were saved in a file *simulationData.csv* having 260 cells (48 empty), 26 rows and ten

Table 8.2: Testing data for the case study device at five rotational speeds. "•" indicates no data collected, corresponding to empty cells of the spreadsheet

2,000 RPM		3,000 RPM		4,000 RPM		5,000 RPM		6,000 RPM	
LPM	Watts	LPM	Watts	LPM	Watts	LPM	Watts	LPM	Watts
1.00	0.1329	1.00	0.4553	1.00	1.0777	1.00	2.0874	1.00	3.5707
1.25	0.1260	1.25	0.4443	1.25	1.0577	1.25	2.0607	1.25	3.5349
1.50	0.1187	1.50	0.4309	1.50	1.0375	1.50	2.0312	1.50	3.4969
1.75	0.1109	1.75	0.4134	1.75	1.0167	1.75	1.9983	1.75	3.4569
2.00	0.1011	2.00	0.3936	2.00	0.9926	2.00	1.9632	2.00	3.4118
2.25	0.0901	2.25	0.3748	2.25	0.9620	2.25	1.9309	2.25	3.3589
2.50	0.0783	2.50	0.3551	2.50	0.9249	2.50	1.8953	2.50	3.3061
•	•	2.75	0.3333	2.75	0.8870	2.75	1.8510	2.75	3.2636
•	•	3.00	0.3063	3.00	0.8495	3.00	1.7890	3.00	3.2158
•	•	3.25	0.2804	3.25	0.8156	3.25	1.7278	3.25	3.1475
•	•	3.50	0.2521	3.50	0.7767	3.50	1.6631	3.50	3.0645
•	•	3.75	0.2193	3.75	0.7252	3.75	1.6051	3.75	2.9675
•	•	4.00	0.1849	4.00	0.6745	4.00	1.5512	4.00	2.8776
•	•	4.25	0.1464	4.25	0.6269	4.25	1.4889	4.25	2.7827
•	•	•	•	4.50	0.5713	4.50	1.4126	4.50	2.7001
•	•	•	•	4.75	0.5130	4.75	1.3226	4.75	2.6209
•	•	•	•	5.00	0.4532	5.00	1.2517	5.00	2.5298
•	•	•	•	5.25	0.3938	5.25	1.1684	5.25	2.4201
•	•	•	•	5.50	0.3274	5.50	1.0764	5.50	2.2929
•	•	•	•	5.75	0.2559	5.75	0.9845	5.75	2.1848
•	•	•	•	6.00	0.1766	6.00	0.8903	6.00	2.0752
•	•	•	•	•	•	•	•	6.25	1.9386
•	•	•	•	•	•	•	•	6.50	1.7997
•	•	•	•	•	•	•	•	6.75	1.6619
•	•	•	•	•	•	•	•	7.00	1.5329
•	•	•	•	•	•	•	•	7.25	1.3981

columns.

8.2.3 Translation

Translation to SAL was coordinated using three inputs to the MATLAB-based translation function:

1. *simulationData.csv*, the name of the spreadsheet file

2. *2*, the number of columns per slice in *simulationData.csv*

3. *simData*, the desired name of the generated SAL context representing spreadsheet data

This generated a 139-line SAL context named *simData.sal* having five slice Booleans (Appendix). Empty cells represented by "•" in Table 8.2 were replaced with zeros.

The generated *constraints* model is similar to the one from Section 8.2.3, except its name is `simData_Constraints` instead of `example_Constraints` and it has five guarded initializations and transitions instead of two.

8.2.4 Additional SAL Model Infrastructure

In support of representing continuous device dynamics within model checking analyses, the generated SAL context was modified in two ways:

1. *plant* model infrastructure was encoded using the technique described in Chapter 3, Section 3.2.2

2. The automatically generated *constraints* model was augmented with inputs that are outputs of the *plant* model

A *plant* model was encoded within the automatically generated SAL context using uninterpreted functions and a SAL module having real number outputs of impeller speed, flow augmentation, and power usage. Unlike uninterpreted functions in relational abstraction, these functions do not need to represent differential equations.

Device information from Section 8.2.1 and the graphical representation of power usage data in Fig. 8.1b are utilized to inform the encoding of a *plant* model. As described in Section 8.2.1, power usage in watts for the case study device is a function of blood flow augmentation in LPM and impeller speed in RPM. Therefore, an uninterpreted function having power usage as an output has LPM and RPM as inputs. The graphical representation of power usage data in Fig. 8.1b indicates that each impeller speed produces a differently sloped curve, each corresponding to a function. Therefore, the *plant* model has one uninterpreted function for each impeller speed (five in total).

Each function's name identifies the impeller speed it represents: `2k_RPM`, `3k_RPM`, `4k_RPM`, `5k_RPM`, `6k_RPM`. As described in the previous paragraph, inputs are real numbers corresponding to blood flow augmentation in LPM (`flow`) and impeller speed in RPM (`speed`). The SAL syntax of one such function is shown below.

```
2k_RPM(flow:  REAL, speed:  REAL): [REAL -> BOOLEAN];
```

Leveraging these five uninterpreted functions, *plant* model output variables represent impeller speed, flow augmentation, and power usage within a SAL module named *device* (encoded as shown below and explained in the next paragraph).

```
device:  MODULE =
BEGIN
  OUTPUT: speed, flow, power:  REAL
  INITIALIZATION
    speed = 2000;
    flow  = 1;
  TRANSITION
    speed' IN {2000, 3000, 4000, 5000, 6000};
    flow'  IN {1, 1.25, 1.5, 1.75, 2, 2.25, 2.5, 2.75, 3,
                3.25, 3.5, 3.75, 4, 4.25, 4.5, 4.75, 5, 5.25,
                5.5, 5.75, 6, 6.25, 6.5, 6.75, 7, 7.25};
  [
      speed = 2000 -->
      power' IN 2k_RPM(flow, speed);
    []speed = 3000 -->
      power' IN 3k_RPM(flow, speed);
    []speed = 4000 -->
      power' IN 4k_RPM(flow, speed);
    []speed = 5000 -->
      power' IN 5k_RPM(flow, speed);
    []speed = 6000 -->
      power' IN 6k_RPM(flow, speed);
  ];
END;
```

Since data collected in computational simulations utilize independent measures of impeller speed and flow augmentation, variables representing them are assigned values independently of other variables. Initial values are 2,000 RPM for impeller speed and 1 LPM for flow augmentation. Next-state values are randomly assigned, where an impeller speed could be any of the five speeds represented in simulation data and flow augmentation can be any of the flow rates. When the model checker runs, it will randomly assign an initial power usage value.

Five guarded transition statements determine next-state values of power usage. Each guard incorporates an impeller speed, and if a guard is satisfied the next-state value of power usage is controlled by a corresponding uninterpreted function.

As-is, power usage outputs of the *plant* model could be any real number, positive or negative, since no other information about power usage is provided. This is corrected by updating the automatically generated *constraints* model with the modifications shown below in colored text (explained in the next paragraph).

```
simData_Constraints:  MODULE =
BEGIN
  INPUT speed, flow, power:  REAL
  LOCAL var_1:  REAL
  LOCAL var_1:  REAL
  OUTPUT fitsData:  BOOLEAN
  DEFINITION
    var_1 = flow;
    var_2 = power;
  INITIALIZATION [
    slice_1(var_1, var_2)   AND speed = 2000  -->
      fitsData = true;
          ...
    []slice_5(var_1, var_2) AND speed = 6000 -->
      fitsData = true;
    []ELSE -->
      fitsData = false;
  ];
  TRANSITION [
    not(slice_1(var_1', var_2'))   AND speed' = 2000 -->
      fitsData' = false;
          ...
    []not(slice_5(var_1', var_2')) AND speed' = 6000 -->
      fitsData' = false;
    []ELSE -->
  ];
END;
```

The first modification (in red) incorporates inputs from the *plant* model representing impeller speed, flow augmentation, and power usage. The second modification (in blue) makes automatically generated local variables `var_1` and `var_2` equal to the corresponding *plant* model inputs. Since slice Booleans represent data with flow in the first column and power usage in the second, `var_1` is made equal to `flow` and `var_2` is made equal to `power`.

The fourth modification (in orange) employs impeller speed inputs within each automatically

generated guard expression. Since each slice Boolean corresponds to data collected for an impeller speed, each automatically generated guard expression is modified by adding a corresponding speed.

For brevity, the first and last guarded initialization and transition commands are shown and an ellipsis (...) is inserted in place of similarly encoded commands.

Table 8.3: LTL Specifications and model checking verification times

LTL syntax	Meaning	Verification time (s)
$\mathsf{G}(fitsData \Rightarrow power > 0)$	If *fitsData* is true, then power usage is always greater than 0 watts	5.07
$\mathsf{G}\begin{pmatrix} fitsData \wedge speed = 2000 \Rightarrow \\ slice_1(flow, power) \end{pmatrix}$	If *fitsData* is true and impeller speed is 2,000 RPM, then flow augmentation and power usage always match the first slice of spreadsheet data	0.33
$\mathsf{G}\begin{pmatrix} fitsData \wedge speed = 3000 \Rightarrow \\ slice_2(flow, power) \end{pmatrix}$	If *fitsData* is true and impeller speed is 3,000 RPM, then flow augmentation and power usage always match the second slice of spreadsheet data	0.37
$\mathsf{G}\begin{pmatrix} fitsData \wedge speed = 4000 \Rightarrow \\ slice_3(flow, power) \end{pmatrix}$	If *fitsData* is true and impeller speed is 4,000 RPM, then flow augmentation and power usage always match the third slice of spreadsheet data	0.39
$\mathsf{G}\begin{pmatrix} fitsData \wedge speed = 5000 \Rightarrow \\ slice_4(flow, power) \end{pmatrix}$	If *fitsData* is true and impeller speed is 5,000 RPM, then flow augmentation and power usage always match the fourth slice of spreadsheet data	0.43
$\mathsf{G}\begin{pmatrix} fitsData \wedge speed = 6000 \Rightarrow \\ slice_5(flow, power) \end{pmatrix}$	If *fitsData* is true and impeller speed is 6,000 RPM, then flow augmentation and power usage always match the fifth slice of spreadsheet data	0.46

8.2.5 Verification

Model checking analyses were conducted to verify that the continuous device model could be utilized to conduct model checking analyses that only consider *plant* model outputs matching simulation data (i.e. the model is correct). To enable such analyses, *simData_Constraints* and *plant* models were composed synchronously to establish a continuous device model using the architecture described in Section 8.1.1.

Six LTL specifications (Table 8.3) were encoded. The first specification of Table 8.3 is needed to ensure that empty cells replaced with zeros by the translation tool are not erroneously considered to fit spreadsheet data, since all dependent measures of power usage computed in simulations were greater than zero. The next five specifications are needed to ensure that each speed setting always has corresponding flow and power values that match spreadsheet data. If no counterexamples are returned for all six specifications, then the continuous device model is considered correct. Specifications were verified using SAL-INF-BMC [68]. All verification reports returned *no counterexample*, indicating that the model is correct. Verification times are shown in the last column of Table 8.3.

8.3 Scalability

To evaluate scalability of model checking analyses employing the MATLAB-based translation tool, two sets of tests were conducting utilizing a modified version of the MATLAB-based tool tool (described later in this section). A set of nine increasingly larger spreadsheet files were generated in MATLAB, and SAL models and specifications were automatically generated. Model checking analyses were conducted using SAL-INF-BMC [68].

Spreadsheet files were generated using the MATLAB function *magic(n)*, which creates a square matrix having n rows, n columns, and n^2 cells containing numeric values. The first matrix had two rows, two columns, and four cells. Subsequent matrices double in size (i.e., the 9th one had 512 rows, 512 columns, and 262,144 cells). Each matrix was then converted to a .csv file. Both sets of tests utilized the same nine files, but SAL contexts were generated using different MATLAB function inputs specifying the number of columns per slice. The notation $r \times c \times s$ is utilized to describe the spreadsheet data organization considered in each automatically generated model, where r is the number of rows, c is the number of columns, and s is the number of slices.

A set of nine SAL contexts and a specification within each context were generated to evaluate scalability with respect to increasingly larger datasets having one slice (i.e. the first file organization is $2 \times two \times 1$ and the last $512 \times 512 \times 1$). Corresponding SAL syntax for each file has one slice Boolean and a *constraints* model having one local variable for each column (i.e. the first SAL context has two local variables and the 9th has 512).

The MATLAB-based translation tool was modified to automatically generate an LTL specification within each SAL context named *one_slice*, which ensures that all spreadsheet cells are considered within model checking analyses. The SAL syntax of each specification is encoded generally below, which reads, "if `fitsData` is true, then the slice Boolean is always true for inputs `var_1,..., var_c`," where the inputted spreadsheet file has c columns within one slice. Automatically generated *constraints* model infrastructure ensures that SAL-INF-BMC always returns *no counterexamples*.

```
one_slice:  THEOREM simData_Constraints |-
                 G(fitsData => slice_1(var_1,..., var_c));
```

Nine more SAL contexts were generated to evaluate scalability with respect to the maximum number of slices, which is c for a spreadsheet file having c columns. The same nine randomly generated spreadsheet files were utilized, but inputs to the translation tool specified that each column is contained within its own slice (i.e. the first file organization is considered $2 \times 2 \times 2$ and the ninth $512 \times 512 \times 512$). The tool was modified to automatically generate an LTL specification named *many_slices*, which has formal semantics ensuring that all slices are considered in model checking analyses. Its SAL syntax is encoded generally below, which reads "if `fitsData` is true, then at least one slice Boolean (`slice_1,..., slice_s`) is always true for inputs `var_1,..., var_c`," where the inputted spreadsheet file has c columns within s slices, and $c = s$. Like *one_slice* specifications, automatically generated *constraints* model infrastructure ensures that SAL-INF-BMC always returns *no counterexamples*.

```
many_slices:  THEOREM simData_Constraints |-
                   G(fitsData => slice_1(var_1,... var_c) OR;
                                 slice_2(var_1,... var_c) OR
                                                ...
                                 slice_s(var_1,... var_c));
```

Verification times for both sets of test are presented in Table 8.4. Verification times increased with model size in both sets of tests; however, increases were larger in models having many slices. Model checking analyses failed in SAL contexts representing spreadsheet data with 262,144 cells, which were too large for SAL-INF-BMC to compute on the target workstation.

Table 8.4: Model checking results for scalability tests. Both tests for a single model are reported in the same row. "—" in the verification time column indicates that model checking failed

Number of cells in spreadsheet	Rows, columns, slices ($r \times c \times s$)	Specification name	Verification time (s)
4	$2 \times 2 \times 1$	*one_slice*	0.01
	$2 \times 2 \times 2$	*many_slices*	0.02
16	$4 \times four \times 1$	*one_slice*	0.04
	$4 \times four \times 4$	*many_slices*	0.02
64	$8 \times 8 \times 1$	*one_slice*	0.12
	$8 \times 8 \times 8$	*many_slices*	0.05
256	$16 \times 16 \times 1$	*one_slice*	0.75
	$16 \times 16 \times 16$	*many_slices*	1.43
1,024	$32 \times 32 \times 1$	*one_slice*	2.57
	$32 \times 32 \times 32$	*many_slices*	4.75
4,096	$64 \times 64 \times 1$	*one_slice*	18.89
	$64 \times 64 \times 64$	*many_slices*	125.76
16,384	$128 \times 128 \times 1$	*one_slice*	513.57
	$128 \times 128 \times 128$	*many_slices*	2,834.29
65,536	$256 \times 256 \times 1$	*one_slice*	2,663.52
	$256 \times 256 \times 256$	*many_slices*	100,382.02
262,144	$512 \times 512 \times 1$	*one_slice*	—
	$512 \times 512 \times 512$	*many_slices*	—

8.4 Discussion

This chapter has presented a novel approach for representing spreadsheet data formally within a continuous device model. A MATLAB-based tool provides an automated way of translating data collected in device tests/simulations to the model checking syntax of SAL. Leveraging a relational abstraction technique, the tool generates formal model infrastructure that could be composed with a separate model of the device, supporting model checking analyses that consider outputs matching simulation data. Applicability of the approach was demonstrated in a case study using computational fluid dynamics simulation data for a medical device under development. Case study results indicate that continuous device models encoded using this approach produce a set of outputs represented in spreadsheet data. Scalability tests indicate that the approach could be utilized to represent spreadsheets having up to 65,536 cells that are organized in up to 256 one-column slices. Considering

these results, the approach shows promise for enabling model checking analyses of continuous device dynamics that are correct with respect to testing/simulation data and scalable with respect to at least 65,536 independent and dependent variable measurements. Methodological considerations and future work are discussed in the following sections.

8.4.1 Methodological Considerations

The approach described in this chapter provides a new way of instantiating the continuous states of a hybrid formalism. Case study results indicate that such a model could be considered correct with respect to medical device simulation data. However, potential applications of such a model were not demonstrated. One way to demonstrate usefulness requires the analyst to incorporate the continuous device model within a larger system model representing discrete device and/or human operator behaviors. Such an application is explored in the next chapter.

While the purpose of the case study was to demonstrate applicability to safety-critical system testing/simulation data, the MATLAB-based tool and relational abstraction-based approach could be applied to any interactive system amenable to testing/simulation of continuous elements, granted that such data are numeric values stored in a .xls, .xlsx, or .csv spreadsheet file. In addition to enabling model checking analyses that incorporate these data, another advantage of the approach is that knowledge of differential equations is not necessary, unlike extant formal methods-based approaches for modeling continuous device elements. Since the MATLAB-based tool automatically generates a *constraints* model, the analyst does not need to manually encode representative relationships among variables within guarded expressions. However, the approach does not preclude relational abstraction: additional *constraints* model infrastructure could be encoded and uninterpreted functions could be represent differential equations if desired.

Scalability tests demonstrated that SAL-INF-BMC supports model checking analyses of data having up to 65,536 cells on a 3.5 GHz workstation with 64 GB RAM running the Ubuntu 14.04 LTS desktop. Organizing each column in its own slice increased verification times substantially in models representing spreadsheets with 4,096 cells or more. This result was surprising: generally, model verification time increases with the number of variables in a formal model (see for example

[166]), and the translation tool generates more variables in models with one slice. The increase in verification times for the second set of tests therefore could have been caused by *many_slices* specifications incorporating more variables than *one_slice* specifications. It could be possible to achieve different scalability results in applied model checking analyses of real testing/simulation data.

8.4.2 Future Work

While knowledge of differential equations is not necessary to apply the approach described in this chapter, some knowledge of both MATLAB and SAL is still required. To remove the need for knowledge of SAL, it could be beneficial to explore ways of automatically generating *plant* model infrastructure (including uninterpreted functions) from spreadsheet data. Another avenue of future work involves a GUI-based model development environment, which could remove the need for knowledge of both MATLAB and SAL. Since the approach is intended for application within a larger, integrated framework, it would be desirable for such a development environment to incorporate the tools and techniques discussed in earlier chapters.

Chapter 9: An Integrated Framework for Verifying Safety-Critical, Human-Interactive System Usability

As discussed in Chapter 1, usability standards and guidelines for safety-critical systems state that the human-system interface should be accurate, understandable, error-tolerant, time-efficient, and complete [17, 23, 27, 20, 29, 19, 24]. Methods from human factors and HCI inform the development of such an interface [88, 131, 221, 89, 82, 90]. Model-based design methodologies at the intersection of human factors and formal methods aid in identifying potential problems and design improvements early in the design cycle [40, 42, 96, 56, 97, 98, 49]. Much work in this design space has centered on analyzing safety-critical properties with respect to normative human task behavior, interface displays/controls, and automation [51**?** , 55, 59, 56, 54, 53, 44, 58, 43, 202, 57].

The five methodologies developed and applied in this research have extended the scope of extant tools and techniques at the intersection of human factors and formal methods. New methodologies enable formal modeling, specification, and verification of documentation navigability, procedure usability, hardware configurability, and interface understandability. The approaches were demonstrated in a series of medical device case studies, and they have been building toward the integrated framework outlined in Chapter 1 (Fig. 1.2). The framework is presented in this chapter. An integrated model architecture is developed to represent a broad range of interactions among human-interactive system elements. Ten generalizable temporal logic specifications and a model checking technique are developed to enable formal verification of usability with respect to an instantiated framework model. An implementation of the framework is demonstrated in a case study based on the pediatric blood pump under development from Chapter 8 and the existing blood pump interface from Chapters 5 and 7. By modifying the existing system, a prototype interface is developed for the pediatric blood pump. An implementation of the model architecture is instantiated to represent the prototype system, and model checking analyses are conducted to verify usability-related specifications. Methodological considerations are discussed in Section 9.13, and areas of future work are

addressed in the next chapter.

9.1 Modeling Methodology: Integrated Framework Architecture

In this research, an interface should support the end user in locating pages of documentation having necessary content, understanding what is signified on the device, executing procedural steps, and avoiding erroneous hardware configurations. Verifying these characteristics requires a formal model representing the integrated human-interactive system. The minimal set of elements that should be modeled includes:

- Documentation having:
  - Navigational tools that are needed to support the end user in locating necessary content
  - Procedures that are needed to support the end user in operating and troubleshooting the device
  - Device descriptions that are needed to explain information that is missing from the device
- Configurable cables, power supplies, and input–output connections
- A control unit having displays, widgets, and internal algorithms
- Actuators connected to the control unit
- End user-device interaction that is shaped by:
  - Knowledge provided by the interface (including procedures, signifiers on the device, and signifiers explained in documentation)
  - Motor capabilities and constraints imposed by the spatial environment (i.e., Gibsonian affordances)

Because there many ways in which these elements could interact, an integrated framework should provide a model architecture enabling the exchange of input/output variables, and the analyst should be able to instantiate the framework using all of its constituent models or a subset. The minimal set of requirements listed below identifies the capabilities of a model architecture enabling such analyses.

1. The architecture should be capable of integrating documentation within a system model

In Chapter 4, a subset of documentation navigation tasks was modeled utilizing integers to represent the pages of a printed or electronic document. The content on each page was not explicitly modeled. In Chapters 5 and 7, procedures and signifiers explained in documentation were modeled independently of page numbers. For the integrated human-system interface to be usable, the end user should be able to locate the pages containing necessary content, and the content on each page should be usable. Thus, to support the analyst in verifying that documentation navigation and the content therein interact in a way that supports usability, the model architecture should integrate documentation navigation, task, and signifier models. The architecture should also enable the analyst to employ the techniques from Chapters 5 and 7 independently of documentation navigation. Thus, the use of a page number output as an input to task and signifier models should be optional.

2. The architecture should be capable of integrating signifiers within a human-system interface model

In Chapter 7, the capabilities of existing frameworks were extended by enabling the formal modeling of visual-, audible-, haptic-, and documentation-channel signifiers; however, variables representing what is signified were not considered with respect to end-user task behavior. To analyze the ways in which signifiers shape task behavior, the analyst may want to utilize variables representing what is signified as inputs to a task model controlling activity execution conditions. Therefore, the framework should be capable of utilizing outputs of a signifier model as inputs to one or more task models.

3. The model architecture should be capable of integrating affordances within a human-system interface model

In Chapter 6, affordances were modeled independently of the end user's goal-driven task behaviors. End user-device interaction was abstracted utilizing the exchange of input/output variables between affordance and human-environment system (HES) models. In Chapters 5 and 7, hardware configurations, displays, control logic, and end user-device interaction were represented without

modeling spatial relations among HES entities. All of these elements are needed within an integrated model of the human-system interface. Thus, to integrate affordances, the architecture should provide a way of exchanging input/output variables among affordance, HES, and end user-device interaction models.

4. The architecture should be capable of integrating controlled actuators within a human-system interface model

Researchers have utilized differential equation abstractions to model the interactions between a target system's control logic and approximate actuator behaviors (see for example [46]). In Chapter 8, a formal model of controlled actuator behaviors was derived from simulation data without the need for approximation or differential equations, but the target system's interface was not modeled. The analyst may want to integrate such a model with models representing the interface. For the interface to govern actuator behaviors, hardware must be configured in a way that supports functionality, and existing methodologies are limited with respect to configurable hardware. For example, a power supply may need to be connected to a control unit for controlled actuators to operate normally. The analyst may want to ensure that the human-system interface supports the end user in configuring hardware in such a way that supports normal actuator operation. Therefore, the architecture should be capable of exchanging input/output variables among display/control logic, actuator, and end user-device interaction models.

5. The architecture should support existing modeling methodologies

As discussed in Chapter 1 researchers have developed a variety of frameworks that support the modeling and verification of human-interactive system usability. As mentioned, the integrated framework should support them. Considering the needs identified in requirements 1–4 (listed above), support for existing methodologies introduces additional input/output variable exchange mechanisms. The minimal set that the architecture should enable is listed below.

5.1. The architecture should support the modeling of interactions between device states and end-user task behaviors

To control what actions execute in a formal task model, researchers have utilized input/output variables representing a variety of elements, including input–output cable connections (see for example [143]), procedural/strategic knowledge (see for example [10]), and end-user interpretation of graphical display messages (see for example [61]). Thus, to support the analyst in utilizing an existing modeling methodology, the architecture should enable the exchange of input/output variables between device and task models.

5.2. The architecture should support the modeling of interactions between affordances and end-user task behaviors

Leveraging task analytic methods, researchers have modeled end-user task behaviors that are shaped by affordances; for example, in [202], a driver's goal of exiting a multi-vehicle highway system was considered reachable if affordances enabled a necessary sequence of driving tasks. Here, an action was allowed to execute only if the affordance enabling it was perceived first (i.e., the affordance needs to exist before an action is attempted). Such a technique was not employed in Chapter 6. To support a similar kind of analysis within the integrated framework, the architecture should enable the exchange of input/output variables between task and affordance models.

The model architecture developed in this research (Fig. 9.1) aims to satisfy the minimal set of requirements. To represent end-user task behaviors, the analyst could encode:

(a) One *documentation navigation* model (Fig. 9.1a) representing an end user navigating through a printed or electronic documentation using its navigational tools

(b) One or more *task* models (Fig. 9.1b), each representing the end user executing a respective procedure. Such a model can be encoded using the technique from Chapter 5 or a technique developed by other researchers (e.g., [56])

An asynchronous composition of these models abstracts documentation navigation tasks and tasks prescribed in documented procedures as separate elements, where the end user could interact with either the document or the device, but not both at the same time. Composing these models syn-

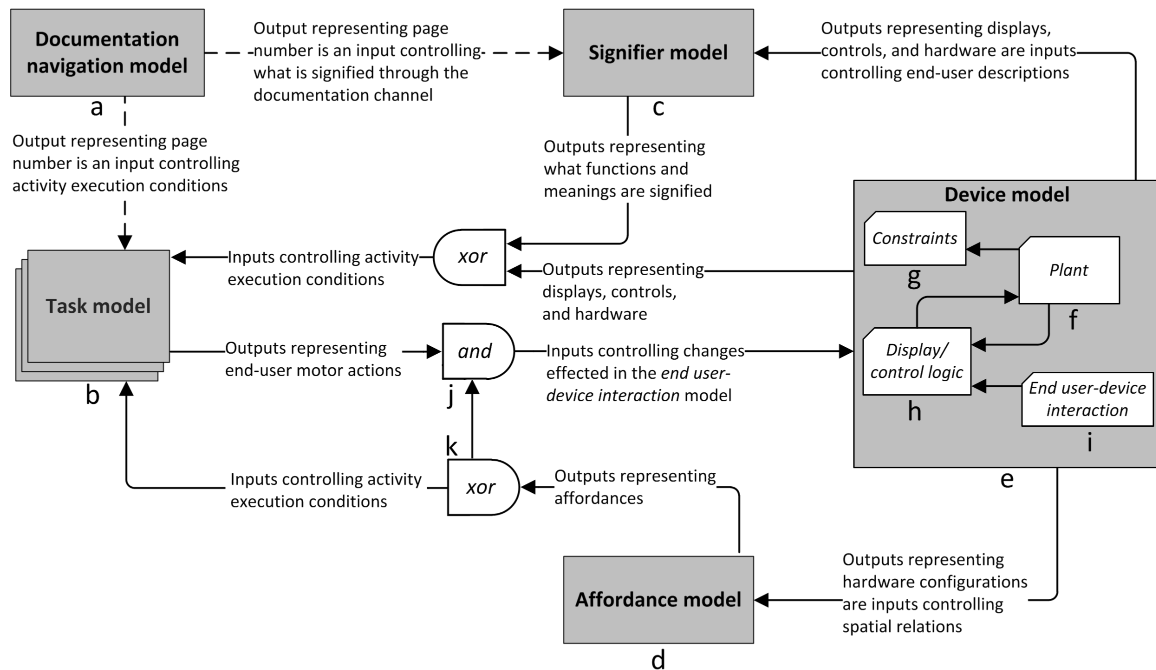


Figure 9.1: Diagrammatic representation of the model architecture employed in the holistic human-system interface framework. Outgoing arrows direct output variables from one model as inputs to another. Labels within each arrow identify what variables are exchanged and what behaviors are controlled. Arrows connecting to shapes labeled *xor* indicate that the analyst must implement the framework using exactly one set of variables identified in the labels of incoming arrows (j) or outgoing arrow (k). Arrows connecting to the shape labeled *and* indicate that the analyst must implement the framework using both sets of variables connected to the shape. Arrows having dotted lines indicate that the use of a page number output of the *documentation navigation* model is optional

chronously represents the end user interacting with both the document and device at the same time, which could be necessary if electronic documentation is incorporated within the device (e.g., a "help" menu on a display screen).

To represent the device (Fig. 9.1e), the analyst could encode four concurrently transitioning models:

(f) A *plant* model representing actuators controlled by the interface (Fig. 9.1f)

(g) A *constraints* model that determines what actuator behaviors are considered in model checking analyses (Fig. 9.1g)

(h) A *display/control logic* model representing the target system's algorithms and what is displayed through visual, audible, and haptic channels (Fig. 9.1h)

(i) An *end user-device interaction* model representing end-user inputs to widgets and configurable hardware (Fig. 9.1i)

Such a composition allows the device model to reflect the current-states of actuators, control logic, and hardware configurations. To integrate device and end-user models, the analyst can encode:

(c) A *signifier model* (Fig. 9.1c) representing what is signified to the end user by visual, audible, and haptic properties of the device, including explanations provided in accompanying documentation

(d) An *affordance model* representing opportunities for action that emerge in the operational environment (Fig. 9.1d)

Composing these models synchronously with *device* models ensures that what is signified and what actions are afforded reflect the current states of displays, controlled actuators, and configurable hardware in the spatial environment.

Depending on what interactions the analyst would like to model, the framework enables a variety of input/output variable exchange combinations. Utilizing a *documentation navigation* model, the analyst could optionally encode the page number output as an input to:

- The *signifier* model such that signifiers operating through the documentation channel depend on what page(s) have been navigated to
- A *task* model such that activity execution conditions depend on what page(s) have been navigated to

One way of using these input/output variable exchange mechanisms treats pages of accompanying documentation similarly to screens of a separate graphical display, where the content on a page (represented as a number) is analogous to the content on a screen (commonly represented using named variable, e.g. "*XrayData*," "*NoData*" [56]). The analyst could instantiate a documentation-integrated model in one of two ways:

1. Documentation-channel signifiers and execution of procedural steps in documentation depend on the current page

2. Documentation-channel signifiers and execution of procedural steps in documentation depend on whether a referent page has been visited

The first technique abstracts a situation in which the end user does not learn what content is in the document; thus, the current-page must contain the procedural steps execute, and a device description must be on the current-page for a function or meaning to be signified through the documentation channel (demonstrated in Section 9.4). The second technique abstracts a situation in which the end user learns what content is on a page after navigating to it once (demonstrated in Appendix C.3).

Utilizing one or more *task* models, the analyst can leverage human action output variables to control next-states of configurable hardware/widgets represented in the *end user-device interaction* model (as in Chapters 5 and 7). Affordances can be integrated in one of two ways:

1. Leveraging a technique developed by other researchers [202], outputs of the *affordance* model can operate as inputs to one or more *task* models controlling activity execution conditions. For example, if a *task* model represents the end user connecting a cable output end to an input socket, a precondition for this action could be that the corresponding connectability affordance is available

2. Using a new technique, *affordance* model outputs can be used in conjunction with human action *task* model outputs to control what changes occur in the *end user-device interaction* model. For example, if an action executes (e.g. the end user attempts to connect a cable output end to an input socket), but the corresponding affordance is not available (e.g. the cable is not connectable for the end user), no change will occur in the end user-device interaction model

To control changes that emerge when an action executes successfully (i.e., the affordance enabling it was available), outputs of the *end user-device interaction* model operate as inputs to the *affordance* model controlling changes to spatial relations among HES entities.

The analyst can integrate signifiers using one of two techniques:

1. The methodology developed in Chapter 7

2. A new methodology integrating *task* and *signifier* models: *signifier* model outputs representing

what is signified can be utilized as inputs to one or more *task* models controlling activity execution conditions. For example, if an alarm is engaged on a device controller, the end user will only begin executing a troubleshooting procedure if the meaning of that alarm is signified through at least one channel

Outputs of the *device* model represent electrical/mechanical interface components (i.e., *end user-device interaction* and *display/control logic* model outputs). As in Chapter 7, they operate as inputs controlling end-user descriptions of visual, audible, and haptic properties of the device; and as in Chapter 5, the same outputs can be leveraged to control activity execution conditions in one or more *task* models. To combine *signifier*, *device*, and *task* models in the integrated architecture, the analyst could utilize different combinations of output variables representing hardware configurations, what is displayed, and what is signified to control activity execution conditions.

Discrete and continuous states are communicated via the exchange of input/output variables between the *display/control logic* and *plant* models. To extend the capabilities of existing frameworks, outputs of the *end user-device interaction* model also operate as inputs to the *display/control logic* model. Such a technique enables the analyst to represent the set of hardware configurations that supports normal actuator functionality (such as a power supply connected to the control unit and a driveline cable connected to a motor). As in Chapter 8, the *constraints* model provides a Boolean output variable enabling model checking analyses that are constrained to a subset of continuous actuator states. For broader applicability, *plant* and *constraints* models can be encoded using either:

- The methodology from Chapter 8, which constrains model checking analyses to continuous states represented in spreadsheet data
- A differential equation abstraction technique (e.g. [46])

9.2 Verification Methodology: Integrated Specifications and Model Checking Technique

The analyses conducted in prior chapters have generally involved no more than two models at a time (one of the user and one of the device/spatial environment). Additionally, specifications have captured no more than one usability-related measure at a time. Considering interactions that can be modeled in the integrated framework architecture, new specifications are needed. One way to derive new specifications involves intersecting safety-critical system usability measures of accuracy, understandability, error tolerance, time efficiency, and completeness (Table 1.3).

Each specification described in Table 1.3 encompasses two measures of safety-critical system usability and characteristics of the interface supporting them. In this research, they are encoded using instantiated model variables and the formal semantics of linear temporal logic (LTL) [62]. To support the analyst in conducting model checking analyses, generalizable LTL syntax for encoding each specification is discussed in the next section.

9.2.1 Specifications

Specifications that are applicable to an integrated framework model can be encoded using propositional operators, the LTL temporal operators "G" meaning "always" and "X" meaning "in the next-state," and the formal semantics of Z [153]: "$\forall$" meaning "for all," "$\exists$" meaning "there exists," and "$\bullet$" meaning "such that." In this section, subscripts i and j (e.g. C_i, C_j) identify unique variables having the same symbol or keyword (as in Chapter 7). For example, leveraging BIGSIS formalism semantics, C_i and C_j represent two distinct interface components; however, in $c_i.vis.s_i$, which represents all visually perceivable properties of a component signifying the same category of function or meaning, the "i" in c_i and the "i" in s_i are unrelated. Leveraging these semantics, generalizable encodings of each specification intersect two safety-critical system usability measures. When these specifications are verified, a counterexample reflects a trace through the model leading up to a potentially unsafe state; otherwise, if no counterexamples are returned, the human-system interface can be considered usable with respect to the specification.

Specifications involving accuracy consider the subset of continuous states fitting spreadsheet

data, a subset of discrete device states, and the end-user actions that execute while following a procedure. Variables representing these states are encoded conjunctively on the left-hand side of an implication operator ($\Rightarrow$). The conjunction of three axioms specifies a subset of states considered in the analysis:

- The Boolean variable *fitsData* (leveraged from the *constraints* model) constrains the analysis to continuous states fitting spreadsheet data
- The axiom $aProcedure = Executing$ (leveraged from EOFM semantics [10] utilized in *task* models) constrains the analysis to end-user actions that execute as specified in a *task* model
- The axiom $state_variable = value_i$ (leveraged from either the *display/control logic* or *end user-device interaction* model) constrains the analysis to a subset of device modes or configurations

Axioms encoded on the right-hand side of the implication operator, denoted by "..." in (9.1), encompass the usability-related characteristics that are desired with respect to constraints specified on the left-hand side.

$$\mathsf{G}(fitsData \wedge aProcedure = Executing \wedge state_variable = value_i \Rightarrow \ldots) \tag{9.1}$$

Specifications involving understandability consider signifier consistency through visual, audible, and haptic channels. If it is relevant, signifier redundancy can also be incorporated within these specifications. For example, redundancy through visual/audible channels may not be relevant for states in which no alarms are engaged and the system is functioning normally (i.e., all relevant signifiers operate through visual, haptic, or documentation channels). However, it may be relevant if there are concurrent visual, audible, and haptic alerts that engage when a malfunction occurs. In these specifications, the formal semantics of consistency (and redundancy, if applicable) assertions are leveraged from Chapter 7. They should be encoded on the right-hand side of the implication operator ($\Rightarrow$, meaning "implies"), and analyses should be constrained to continuous states fitting spreadsheet data. This ensures that the model checker only consider signifiers with respect to states enabled in the *continuous device* model.

Specifications involving error-tolerance consider:

- One or more affordances that should not emerge when the system is in a particular configuration (as in the *weak affordance error tolerance* specification of Chapter 6), or
- One or more affordances that should never emerge in any configuration, such as the opportunity to connect a cable output end to the wrong input socket (as in the *strong affordance error tolerance* specification of Chapter 6)

Axioms encoded on the right-hand side of the implication operator, denoted by "$affordance_{unsafe}$" in (9.2), encompass one or more affordances that should not emerge when constraints on the left-hand side (denoted by "$\ldots$") are satisfied.

$$\mathsf{G}(\ldots \Rightarrow \neg affordance_{unsafe}) \tag{9.2}$$

Specifications involving time efficiency consider a discrete device state that emerges in the immediate next-state, inspired by the display/control feedback specifications developed at the intersection of formal methods and human factors [58, 43]. Axioms encoded on the left-hand side of the implication operator in (9.3) encompass one or more discrete device states (denoted by $state_variable = value_i$) that change in the next state (denoted by $\mathsf{X}(state_variable = value_j)$). Axioms encoded on the left-hand side (denoted by "$\ldots$") should be satisfied.

$$\mathsf{G}(fitsData \wedge state_variable = value_i \wedge \mathsf{X}(fitsData \wedge state_variable = value_j) \Rightarrow \ldots) \tag{9.3}$$

Specifications involving completeness involve a function or meaning signified through all channels. In (9.4), axioms encoded on the left-hand side of the implication operator always include the variable *fitsData*. This ensures that model checking analyses are constrained to continuous variables fitting simulation data. Depending on what is encoded in place of "$\ldots$" on the left-hand side, an axiom encoded on the right-hand side could assert one of two things:

1. The word corresponding to nothing signified (w_0) is not signified through all channels (corre-

sponding to the *signifier completeness* specification of Chapter 7)

2. A particular function or meaning (w_i) is signified through at least one channel (utilized for the *accuracy and completeness* specification explained later in this section)

In (9.4), a vertical bar separating these two kinds of axioms denotes that the analyst could encode one or the other exclusively.

$$\mathsf{G}\left(\mathit{fitsData} \wedge \ldots \Rightarrow \neg \begin{pmatrix} \mathit{visually_signified}_i = w_0 \wedge \\ \mathit{audibly_signified}_i = w_0 \wedge \\ \mathit{haptically_signified}_i = w_0 \wedge \\ \mathit{documented}_i = w_0 \end{pmatrix} \middle| \begin{matrix} \mathit{visually_signified}_i = w_i \vee \\ \mathit{audibly_signified}_i = w_i \vee \\ \mathit{haptically_signified}_i = w_i \vee \\ \mathit{documented}_i = w_i \end{matrix} \right) \quad (9.4)$$

Beginning with *accuracy and understandability* (Table 1.3, upper-left-hand corner) each of the ten integrated framework specifications are listed, explained, and encoded generally below.

1. *Accuracy and understandability*: signifiers are accurate, consistent, and (optionally) redundant while a procedure is executing and the system is in a particular state

This specification could help ensure that signifiers are understandable (i.e., consistent) and correct (i.e., accurate) with respect to a particular situation, such as when an alarm is engaged and the end user is executing a procedure to correct it. A potential violation of this specification was observed in the Three Mile Island accident: human operators understood what was signified by individual indicator lights on the coolant system control panel; however, multiple, concurrently illuminated lights had conflicting meanings, and the procedure required for corrective action was not executed in a way that could correct the problem [209].

A example specification is shown in (9.5) for a hypothetical framework model having a device model with i state variable values and signifiers operating through visual, audible, and haptic channels. It reads, "it is always true (G) that when a procedure is executing ($\mathit{aProcedure} = \mathit{Executing}$) and certain device conditions are satisfied ($\mathit{fitsData} \wedge \mathit{state_variable} = \mathit{value}_i$), this implies ($\Rightarrow$) that signifiers operating through visual, audible, and haptic channels are consistent (i.e., they all signify

the same thing) and one of them signifies the correct state ($c_{i \vee j}.P_{i \vee j}.s_i = \mathit{signified}_{value_i}$)."

$$\mathsf{G}\left(\begin{array}{lcl} \mathit{aProcedure} = \mathit{Executing} & & C_i.vis.S_i = C_j.vis.S_i \wedge \\ \mathit{fitsData} \wedge & \Rightarrow & C_i.aud.S_i = C_j.aud.S_i \wedge \\ \mathit{state_variable} = \mathit{value}_i \wedge & & C_i.hap.S_i = C_j.hap.S_i \wedge \\ & & C_{i \vee j}.P_{i \vee j}.S_i = \mathit{signified}_{value_i} \end{array}\right) \quad (9.5)$$

2. *Accuracy and error tolerance*: an unsafe affordance does not emerge while a procedure is executing and the system is in a particular state

This specification could help ensure that a procedure and the configurable hardware involved in the procedure are designed in a way that prevents an unsafe affordance. Such a problem was observed in the medical device adverse event analyzed in Chapter 6: during a surgical procedure, the surgeon was able to connect a pacemaker lead output end to the wrong import end of the pulse generator [203]. While the procedure was not modeled in Chapter 6, the integrated framework broadens the analytic scope in a way that could help identify problems involving interactions among the temporal ordering of procedural steps, task descriptions identifying part-whole components, and hardware configurability [204].

An example specification is shown in (9.6) for a hypothetical framework model having a device model with i state variable values. It reads, "it is always true (G) that when a procedure is executing ($\mathit{aProcedure} = \mathit{Executing}$) and certain device conditions are satisfied ($\mathit{fitsData} \wedge \mathit{state_variable} = \mathit{value}_i$), this implies ($\Rightarrow$) that an unsafe affordance does not emerge ($\neg \mathit{affordance}_{unsafe}$)."

$$\mathsf{G}\left(\begin{array}{lcl} \mathit{aProcedure} = \mathit{Executing} \wedge & & \\ \mathit{fitsData} \wedge & \Rightarrow & \neg \mathit{affordance}_{unsafe}) \\ \mathit{state_variable} = \mathit{value}_i & & \end{array}\right) \quad (9.6)$$

For such a specification to be applicable the analyst could encode one or more affordances that

are always unsafe using the CAVEMEN approach, as in Chapter 6. Alternatively, affordances that are safe in some situations may be unsafe in others, such as aircraft fuselage door openability when the aircraft is above a certain altitude (i.e., door openability is unsafe) or on the ground (i.e., door openability is safe). Such an affordance can be encoded on the right-hand side of the implication operator for a particular situation represented on the left. This technique is demonstrated in Section 9.11.

3. *Accuracy and time efficiency*: if a procedure is executing and the next-state of the device is different from the current-state, a desired affordance emerges in the next-state

As mentioned in Chapter 6, the analyst may want to ensure that configurable hardware enables an affordance to emerge at the right time, such as immediately after a particular change-of-state for the device. Consider the home-use dialysis machine and setup procedure mentioned in Chapter 5: during setup, patients should connect a fluid-filled tube after it has been properly primed; thus, in support of accuracy and time efficiency, the affordance of "tube connectable" should emerge when the fluid-filled tube transitions to "primed." A specification enabling such an analysis is encoded generally in (9.7). It reads, "it is always true (G) that when a procedure is executing ($aProcedure_i = Executing$), certain device conditions are satisfied ($fitsData \wedge state_variable = value_i$), and these conditions change in the next-state ($\mathsf{X}(fitsData \wedge state_variable = value_i)$), this implies ($\Rightarrow$) that a desired affordances emerges in the next-state ($\mathsf{X}(affordance_{safe})$)."

$$\mathsf{G}\left(\begin{array}{l} aProcedure_i = Executing \wedge \\ fitsData \wedge \\ state_variable = value_i \wedge \\ \mathsf{X}(fitsData \wedge state_variable = value_j) \end{array} \Rightarrow \mathsf{X}(affordance_{safe})\right) \tag{9.7}$$

4. *Accuracy and completeness*: signifiers are accurate and complete while a procedure is executing and the device is in a particular state

This specification could help ensure that information that while a procedure is executing, relevant

functions/meanings are signified to the end user through at least one channel. A potential violation of this specification was observed in the medical device adverse event discussed in Chapter 7: a patient went into cardiac arrest during surgery, and a surgical team member reacted by pushing a red button labeled "stop" on the X-ray imaging system control panel. This action unexpectedly shut down the life support system, and the patient expired [207]. Visual signifiers on the button may have been insufficient for understanding the consequences of pushing it; additionally, technical documentation on the manufacturer's website does not explain what is meant by the red color and "stop" label [208]. Thus, this event can be characterized as a violation of signifier completeness and accuracy with respect to a particular procedure and signified function.

An example specification is shown in (9.8) for a hypothetical framework model having a device model with j state variables and signifiers operating through visual, audible, haptic, and documentation channels. It reads, "it is always true (G) that when a procedure is executing ($aProcedure = Executing$) and certain device conditions are satisfied ($fitsData \wedge state_variable = value_i$) , this implies ($\Rightarrow$) that a function or meaning corresponding to $value_i$ is signified through at least one channel."

$$\mathsf{G}\left(\begin{array}{lcl} & & visually_signified_i = w_{value_i} \vee \\ aProcedure = Executing \wedge & & \\ & & audibly_signified_i = w_{value_i} \vee \\ fitsData \wedge & \Rightarrow & \\ & & haptically_signified_i = w_{value_i} \vee \\ state_variable = value_i & & \\ & & documented_i = w_{value_i} \end{array}\right) \tag{9.8}$$

5. *Understandability and error tolerance*: an unsafe affordance does not emerge and signifiers are consistent and (optionally) redundant

This specification considers simultaneous error-tolerance and understandability by combining concepts from Chapters 6 and 7. Consider the adverse event involving the red button labeled "stop" on the X-ray imaging system control panel: as mentioned, the color and label of the button could have been insufficient for understanding the consequences of pushing it. Alternatively, the red color and

"stop" label could have had conflicting (i.e., inconsistent) meanings, perhaps signifying "emergency" and "shut down the system" respectively. In either case, the event could have been prevented if the affordance of button pushability did not exist at an inappropriate time. These problems reflect concurrent violations of understandability, error tolerance, or both.

A specification asserting the concurrent absence of such problems is shown in (9.9) for a hypothetical framework model having one unsafe affordance encoded in the *affordance* model and signifiers operating through visual, audible, and haptic channels encoded in the *signifier* model. It reads,

- "It is always true (G) that an unsafe affordance never emerges ($\neg(\mathit{affordance}_{unsafe})$); and ($\wedge$),
- it is always true (G) that when certain device conditions are satisfied ($\mathit{fitsData} \wedge \mathit{state_variable} = \mathit{value}_i$), this implies $\Rightarrow$ that signifiers operating through all channels are consistent for all categories of function/meaning"

$$\mathsf{G}(\neg \mathit{affordance}_{unsafe}) \wedge \mathsf{G}\left(\mathit{fitsData} \Rightarrow \begin{array}{c} \forall\, c_i : C_i;\ c_j : C_j;\ s_i : S_i \bullet \\ c_i.vis.s_i = c_j.vis.s_i \wedge \\ c_i.aud.s_i = c_j.aud.s_i \wedge \\ c_i.hap.s_i = c_j.hap.s_i \end{array} \right) \tag{9.9}$$

6. *Understandability and time efficiency*: if the next-state of the system is different from the current-state, signifiers are consistent and (optionally) redundant in the next-state

This specification combines signifier understandability specifications developed in Chapter 7 with feedback specifications developed by other researchers [58, 59]. It could help ensure that signifiers are updated in a way that is both time-efficient and understandable. An example specification is shown in (9.10) for a hypothetical framework model having a *discrete device* model with j modes and signifiers operating through visual, audible, and haptic channels encoded in the *signifier* model. It reads, "it is always true (G) that when certain device conditions are satisfied ($\mathit{fitsData} \wedge \mathit{state_variable} = \mathit{value}_i$) and these conditions change in the next-state ($\mathsf{X}(\mathit{fitsData} \wedge \mathit{state_variable} = \mathit{value}_j)$), this

implies ($\Rightarrow$) that, in the next-state (X), signifiers operating through all channels are consistent for all categories of function/meaning."

$$\mathsf{G}\left(\begin{array}{l} \mathit{fitsData} \wedge \\ \mathit{state_variable} = \mathit{value}_i \wedge \\ \mathsf{X}(\mathit{fitsData} \wedge \mathit{state_variable} = \mathit{value}_j) \end{array} \Rightarrow \mathsf{X}\left(\begin{array}{c} \forall\, c_i : C_i;\; c_j : C_j;\; s_i : S_i \bullet \\ c_i.vis.s_i = c_j.vis.s_i \wedge \\ c_i.aud.s_i = c_j.aud.s_i \wedge \\ c_i.hap.s_i = c_j.hap.s_i \end{array}\right)\right) \tag{9.10}$$

7. *Understandability and completeness*: signifiers are consistent, (optionally) redundant, and complete

This specification combines the signifier specifications developed in Chapter 7. It could help ensure that what is signified on the device is understandable with respect to consistency. As mentioned in Chapter 7, it could be possible for a signifier consistency specification to be satisfied spuriously if all perceivable properties operating through the same channel never signify any functions or meanings to the end user (i.e., they are always equal to the word corresponding to nothing signified, w_0). The same can be said of a redundancy specification, where all perceivable properties operating concurrently through visual, audible, and haptic channels always signify nothing. In such cases, it is critical that signifiers operate through the documentation channel for the interface to be complete. The example specification in (9.11) asserts such a property. It reads, "it is always true (G) that when the device is operating within data-constrained parameters (*fitsData*), this implies ($\Rightarrow$) that signifiers operating through all channels are consistent and complete for all categories of function/meaning."

$$\mathsf{G}\left(\begin{array}{l} \\ \mathit{fitsData} \Rightarrow \end{array}\begin{pmatrix} \forall\, c_i : C_i; \\ \quad c_j : C_j; \\ \quad s_i : S_i \bullet \\ \quad c_i.vis.s_i = c_j.vis.s_i \wedge \\ \quad c_i.aud.s_i = c_j.aud.s_i \wedge \\ \quad c_i.hap.s_i = c_j.hap.s_i \end{pmatrix} \wedge \neg \begin{pmatrix} \forall\, \mathit{visually_signified}_i; \\ \quad \mathit{audibly_signified}_i; \\ \quad \mathit{haptically_signified}_i; \\ \quad \mathit{documented}_i : \mathit{signified_i} \bullet \\ \quad \mathit{visually_signified}_i = w_0 \wedge \\ \quad \mathit{audibly_signified}_i = w_0 \wedge \\ \quad \mathit{haptically_signified}_i = w_0 \wedge \\ \quad \mathit{documented}_i = w_0 \end{pmatrix}\right) \tag{9.11}$$

8. *Error tolerance and time efficiency*: if the next-state of the system is different from the current-state, an unsafe affordance does not emerge in the next-state

The error tolerance-related specifications discussed thus far have involved an unsafe affordance that emerges while a procedure is executing. In support of ensuring error tolerance and time efficiency, the analyst may want to verify that an unsafe affordance does not emerge at a particular time, such as immediately after an automated event or human input effects a change in the system. For example, in the case study of Chapter 5, a potential time-efficiency problem involved the end user connecting a lithium-ion battery without checking its charge level first; and in regard to error tolerance, a possibly discharged or malfunctioning battery that was disconnected earlier in the procedure was connectable to the replacement controller. Considering both time efficiency and error tolerance, the analyst may want to verify that a lithium-ion battery becomes connectable to the replacement controller immediately after checking the battery level.

The specification in (9.12) enables such analyses for a hypothetical framework model representing a device with at least one affordance considered unsafe with respect to a particular, temporally ordered change to the device. It reads, "it is always true (G) that when certain device conditions are satisfied ($\mathit{state_variable} = \mathit{value}_i$) and ($\wedge$) these conditions change in the next-state ($\mathsf{X}(\mathit{state_variable} = \mathit{value}_j)$), an unsafe affordance does not emerge in the next-state

$(\mathsf{X}\neg(\mathit{unsafeUnaffordance}))$."

$$\mathsf{G}\left(\begin{array}{l} \mathit{state_variable} = \mathit{value}_i \wedge \\ \mathsf{X}(\mathit{state_variable} = \mathit{value}_j) \end{array} \Rightarrow \mathsf{X}\neg(\mathit{affordance}_{unsafe})\right) \quad (9.12)$$

9. *Error tolerance and completeness*: an unsafe affordance does not emerge and signifiers are complete

This specification combines error tolerance and completeness specifications with respect to one unsafe affordance and signifiers of all functions/meanings operating through all channels. It is similar to the *understandability and error tolerance* specification (9.9), but instead of asserting error tolerance and signifier consistency (and, optionally, redundancy), the specification asserts error tolerance and completeness. Consider the adverse event involving the red button labeled "stop" on the X-ray imaging system control panel: as mentioned, signifiers on the pushbutton may have been inconsistent; alternatively, the system's documentation may not explain the consequences of pushing the button. In either case, the event could have been prevented if the affordance of button pushability did not exist at an inappropriate time. These problems reflect concurrent violations of error-tolerance, completeness, or both.

A specification asserting the concurrent absence of such problems is shown in (9.13) for a hypothetical framework model having one unsafe affordance encoded in the *affordance* model and signifiers operating through visual, audible, haptic, and documentation channels encoded in the *signifier* model. It reads, "it is always true (G) that an unsafe affordance never emerges ($\neg(\mathit{affordance}_{unsafe})$) and ($\wedge$) it is always true ($\mathsf{G}$) that signifiers of all categories of function/meaning are complete."

$$\mathsf{G}\neg(affordance_{unsafe}) \wedge \mathsf{G}\neg\left(\begin{array}{l} fitsData \Rightarrow \\ \quad \forall\, visually_signified_i;\ audibly_signified_i; \\ \quad\quad haptically_signified_i;\ documented_i : \\ \quad signified_i \bullet \\ \quad\quad\quad visually_signified_i = w_0 \wedge \\ \quad\quad\quad audibly_signified_i = w_0 \wedge \\ \quad\quad\quad haptically_signified_i = w_0 \wedge \\ \quad\quad\quad documented_i = w_0 \end{array}\right) \tag{9.13}$$

10. *Time efficiency and completeness*: if the next-state of the system is different from the current-state, signifiers are complete in the next-state

This specification combines the signifier completeness specification developed in Chapter 7 with feedback specifications developed by other researchers [58, 59]. It could help ensure that signifiers are updated in a way that is both time efficient and complete. An example specification is shown in (9.10) for a hypothetical framework model having a *discrete device* model with j state variables and signifiers operating through visual, audible, haptic, and documentation channels encoded in the *signifier* model. It reads, "it is always true (G) that when certain device conditions are satisfied ($fitsData \wedge state_variable = value_i$) and ($\wedge$) these conditions change in the next-state ($\mathsf{X}(fitsData \wedge state_variable = value_j)$), this implies ($\Rightarrow$) that signifiers of all categories of function/meaning are complete in the next-state."

$$\mathsf{G}\left(\begin{array}{l} \mathit{fitsData} \wedge \\ \mathit{state_variable} = \mathit{value}_i \wedge \\ \mathsf{X}(\mathit{state_variable} \neg \mathit{value}_i) \end{array} \Rightarrow \mathsf{X}\neg \left(\begin{array}{l} \forall\ \mathit{visually_signified}_i;\ \mathit{audibly_signified}_i; \\ \quad \mathit{haptically_signified}_i;\ \mathit{documented}_i : \\ \mathit{signified_i} \bullet \\ \quad \mathit{visually_signified}_i = w_0 \wedge \\ \quad \mathit{audibly_signified}_i = w_0 \wedge \\ \quad \mathit{aptically_signified}_i = w_0 \wedge \\ \quad \mathit{documented}_i = w_0 \end{array}\right)\right) \tag{9.14}$$

9.2.2 Model Checking Technique

SAL-INF-BMC is utilized to verify specifications for an integrated framework model; however, using the default settings of SAL-INF-BMC, it can only generate counterexamples having no more than ten steps [68]. If counterexamples having more than ten steps exist, they will not be uncovered unless the analyst increases search depth to accommodate the longest possible counterexample [149]. Increasing search depth of a bounded model checking analysis generally increases verification time; thus, it could be beneficial for the analyst to compute the longest possible counterexample of an integrated framework model before invoking SAL-INF-BMC. This can be accomplished automatically using symbolic model checking. The method utilized in this research proceeds as follows for a integrated framework model and each LTL specification:

1. Invoke SAL-INF-BMC at the default counterexample length of ten. If a counterexample is returned, the system could have a potential usability problem with respect to the specification. If no counterexample is returned:

2. Asynchronously compose *documentation navigation*, *end user-device interaction*, *affordance*, and all *task* models within a new system model: a discrete framework model encompassing a subset of discrete state variables

3. Instantiate the specification encoded generally in (9.15), which reads, "it always untrue ($\mathsf{G}\neg$)

that all procedures are executing and done at the same time." Such a specification is trivially always true, and it ensures that the model checker visits all reachable states

4. Pass this specification and the discrete framework model to SAL-SMC. The verification report will return "proved" and the number of iterations required to prove the specification (i.e., steps that are needed to find the longest possible counterexample). This number is greater than or equal to the depth required for a bounded model checking analysis to be exhaustive with respect to all discrete state variables

5. Pass the integrated framework model and a specification to SAL-INF-BMC at the appropriate depth

$$\mathsf{G}\neg\begin{pmatrix} aProcedure_1 = Done \wedge aProcedure_1 = Executing \\ \wedge \ldots \wedge \\ aProcedure_n = Done \wedge aProcedure_n = Executing \end{pmatrix} \tag{9.15}$$

9.3 Case Study

To evaluate applicability of the integrated framework, a case study was developed based on the early prototype of a safety-critical, human-interactive system. The target system is a pediatric blood pump (actuator dynamics modeled in Chapter 8) and a prototype interface based on the system from Chapters 5 and 7, an LVAD intended for adult patients. The prototype interface includes configurable hardware of the existing system and modified versions of its documentation (a 29-page patient handbook) and controller. Because the case study device is for pediatric patients, end users of the implanted pump are considered too young to be end users of the interface. End users of the interface are assumed to be normally-abled, English-speaking, adult caregivers having no training or experience operating or troubleshooting the device.

Leveraging simulation from Chapter 8, the existing system's displays and controls are modified to fit the pump speed and power supply parameters of the pediatric pump. Utilizing design insights

gleaned in previous chapters:

- The pump stopped alarm troubleshooting procedure from Chapter 5 is modified in a way that aims to improve accuracy, time efficiency, and completeness
- Signifiers from Chapter 7 and the controller's algorithms are modified in a way that aims to improve consistency, redundancy, and completeness while fitting power and speed dynamics of the pediatric pump

Informed by safety-critical system usability measures and design standards:

- A draft patient handbook is developed with navigational tools that aim to support end users is quickly navigating within and between content listed on different pages
- An operational procedure is developed to support end users in adjusting the pump speed to a desired speed

To be usable, this interface should support the end user in locating pages of the patient handbook having necessary content, understanding what is signified, accurately manipulating configurable hardware, executing procedural steps, and avoiding erroneous hardware configurations. To evaluate applicability of the framework, a subset of usability measures (discussed later) are investigated within one possible implementation of the integrated model architecture.

One of each specification from Section 9.2.1 is encoded and verified using the model checking technique developed in Section 9.2.2.

9.3.1 Configurable Hardware

The case study system's hardware components include the controller, cables, connectors, portable lithium-ion batteries, and lead battery considered in Chapter 5. As mentioned in Chapter 5, interface components are intended for portability such that external controllers and batteries may be carried using harnesses, holsters and/or straps. Caregivers must also carry select replacement components at all times in the case of a mechanical malfunction or power supply issues. The components considered in this case study include two controllers (Fig. 9.2a–c), two lithium-ion battery cables (Fig. 9.2d, e),

one pump cable (Fig. 9.2f, g), one abdominal cable (Fig. 9.2h, i), one Y-cable (Fig. 9.2j–l), one lead battery (Fig. 9.2m, n), two lithium-ion batteries (Fig. 9.2o–q), and one red tag (Fig. 9.2r).

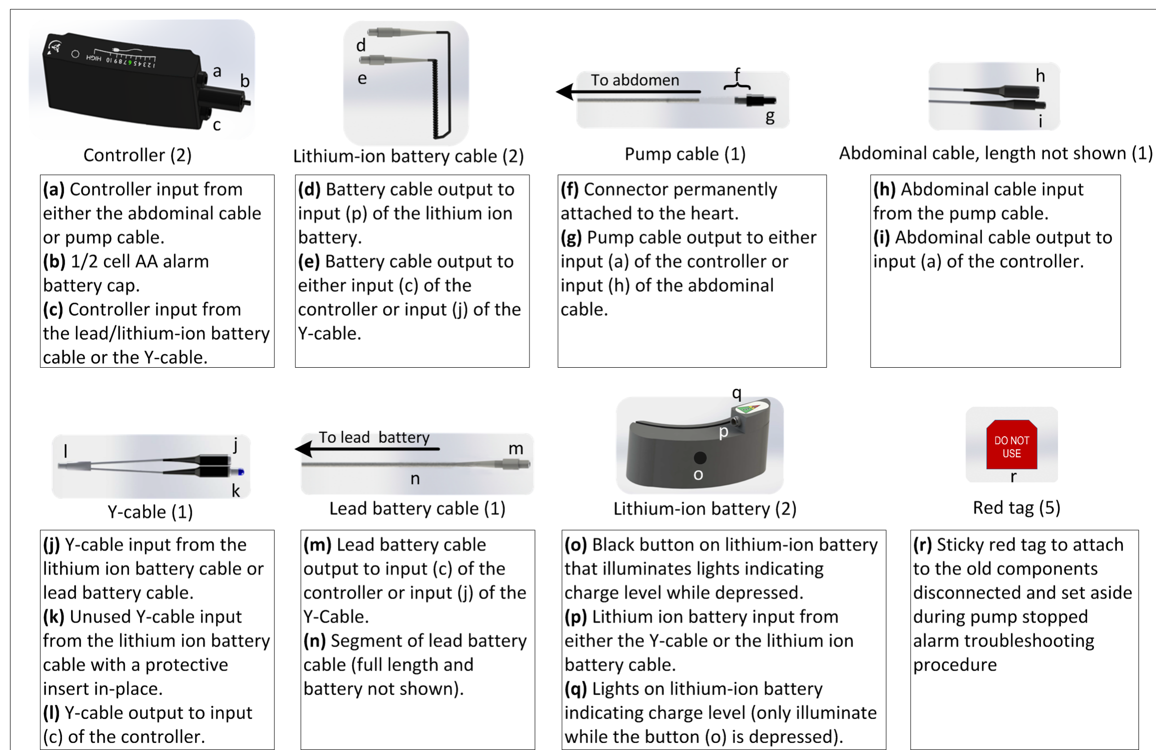


Figure 9.2: Labeled diagram of case study configurable hardware components. This figure appears on pages 13 and 14 of the modified patient handbook to aid end users in identifying components involved in the pump stopped alarm troubleshooting procedure

For the device to function normally, a power supply must be connected to the controller; the pump must be connected to the controller (either via the abdominal cable or directly via the pump cable); and the connector permanently attached to the heart must be assembled. In such a configuration, all components may appear to be functioning normally; and on the system's controller, no alarm could be engaged. The pump stopped alarm or the high power alarm could engage if a malfunction occurs while the system is in a functional configuration. Otherwise, is a power supply/pump cable is not connected or the permanently attached connector is broken, the pump stopped alarm engages. While it is also necessary for the power supply to be charged, battery charge levels are not modeled in this case study.

9.3.2 Modified Controller

The case study controller has four alarms and a speed setting knob having five settings numbered 1–5. The same three components modeled in the case study of Chapter 7 are considered in this case study: the audible/visual pump stopped alarm, the audible/visual high power alarm, and the speed setting knob. Control logic involving visual/audible properties of these components is modified to fit the performance characteristics of the pediatric pump from Chapter 8, while perceivable properties operating as signifiers are modified to support end users in understanding what alarms mean. Like the original controller, the modified one supplies as much power as necessary for the pump to maintain one of five programmed speeds. The relationships between power supply and pump speed were modified to fit simulation data for the case study pump (Table 9.1). An audible alarm can be emitted if the 1/2 cell AA battery cap of a controller is tightened. An internal battery that is separate from the 1/2 cell AA battery powers the pump stopped alarm indicator light when no power supply is connected to the controller.

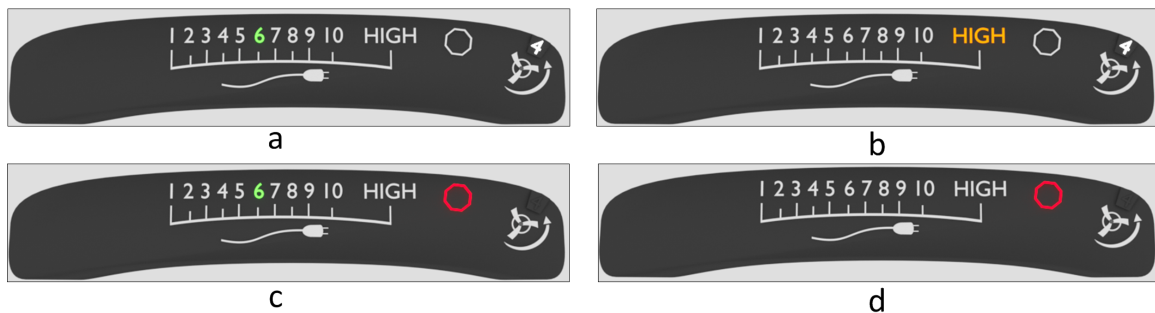


Figure 9.3: Graphical rendering of the case study system's controller. (a) The speed setting number 4 illuminates white and the power indicator light 6 illuminates green. (b) High power alarm engaged: the word "HIGH" illuminates amber and a loud alarm emits the phrase "power too high" with a one-second pause between emissions. (c) Pump stopped alarm engaged while power is still supplied to the pump: the octagon shaped light illuminates red, one power indicator light illuminates green, the speed setting knob light turns off, and a loud alarm emits the phrase "pump stopped" with a one-second pause between emissions. (d) Pump stopped alarm engaged while no power supplied to the pump: the octagon shaped light illuminates red, the speed setting knob light turns off, the power indicator light turns off, and a loud alarm emits the phrase "pump stopped" with a one-second pause between emissions.

Graphical renderings of the modified controller are shown in Fig. 9.3. For the power indicator lights under normal conditions (i.e., no alarm is engaged), any one light numbered 1–10 illuminates green (Fig. 9.3a) to signify power units supplied to the pump (Table 9.1). Each light maps to one of

Table 9.1: Control knob settings, programmed speeds, and approximate power supplied by the controller leveraged from simulation data (Chapter 8)

Power indicator number illuminated	Approximate power range (watts)	Setting	Pump speed (RPM)	Flow range (LPM)
1	0.05–0.15	1 2	2,000 3,000	1.00–2.25 4.25
2	0.16–0.30	2 3	3,000 4,000	3.00–4.00 5.75–6.00
3	0.31–0.45	2 3	3,000 4,000	1.25–3.00 5.00–5.50
4	0.46–0.70	2 3	3,000 4,000	1.00 4.00–4.75
5	0.71–1.00	3 4	4,000 5,000	2.00–3.75 5.75–6.00
6	1.01–1.40	3 4 5	4,000 5,000 6,000	1.00–1.75 4.50–5.50 7.25
7	1.41–1.90	4 5	5,000 6,000	2.50–4.25 6.50–7.00
8	1.91–2.50	4 5	5,000 6,000	1.00–2.25 5.25–6.25
9	2.51–3.10	5	6,000	3.50–5.00
10	3.11–3.60	5	6,000	1.00–3.25

ten non-overlapping power ranges. No lights illuminate when the controller is supplying no power (Fig. 9.3d). This design was implemented because of a potential usability problem uncovered in Chapter 7: the BIGSIS-XML encoding process revealed that power indicator lights numbered 8–9 could signify multiple pump speeds, while the BIGSIS-SAL model encoding process revealed that the numbers 11–12 on the controller never illuminate. In the modified design, all lights are utilized such that each light could signify one pump speed.

For the high power alarm, the indicator light labeled "HIGH" illuminates amber and a loud, periodic alarm emits the phrase "power too high" when the controller is supplying more than 3.5 watts (Fig. 9.3b), where the maximum power supplied is four watts. This high power alarm control logic was chosen to enable a possible high power alarm when the pump is operating within the constraints of the simulation data. The new audible high power alarm was implemented to aid the

end user in distinguishing the high power alarm from the pump stopped alarm, addressing a potential problem uncovered in the Chapter 7 case study (model checking counterexample to *PowerSupplied visual audible redundancy*). Additionally, the indicator light has the word "HIGH" instead of the number 13, which could be an improved representation of the alarm's meaning to an untrained user.

For the pump stopped alarm, the octagon light illuminates red and a loud, periodic alarm emits the words "pump stopped" when the pump speed is 0 RPM (Fig. 9.3c). The new design was implemented to address a potential usability problem identified in Chapter 7, Section 7.5.3: the original controller has properties signifying a pump speed of *ZeroRPM* when the pump stopped alarm is engaged; however, this alarm could engage when the actual pump speed is above 0 and below 5,000 RPM. This concern is addressed by modifying the controller's algorithms so the pump stopped alarm engages when the pump speed is 0 RPM. The new audible alert for this alarm was implemented to aid the user in distinguishing it from the high power alarm, which maps to the same pattern and volume in the original controller.

The speed setting knob has a backlight that turns off when the pump stopped alarm is engaged (as in Fig. 9.3c) and illuminates white when the pump stopped alarm is not engaged (as in Fig. 9.3a). This modification to the original controller was implements to address a potential usability problem identified in Chapter 7: the counterexample to *visual consistency* revealed that the pump stopped alarm and the speed setting knob label could signify two different pump speeds at the same time when the pump stopped alarm is engaged. The end-user description *PumpStoppedAlarm.Color.Colored = Red* only occurs when the pumps topped alarm is engaged, but the speed setting knob label is painted on the knob, producing an end-user description of *SpeedSettingKnob.Label.Labeled = Four*. In the modified design, labels painted on the speed setting knob are replaced with transparent labels that are backlit. When the pump stopped alarm is engaged, the backlight disengages, which could enable an end-user description of *SpeedSettingKnob.Label.Labeled = noLabel*.

9.3.3 Modified Documentation

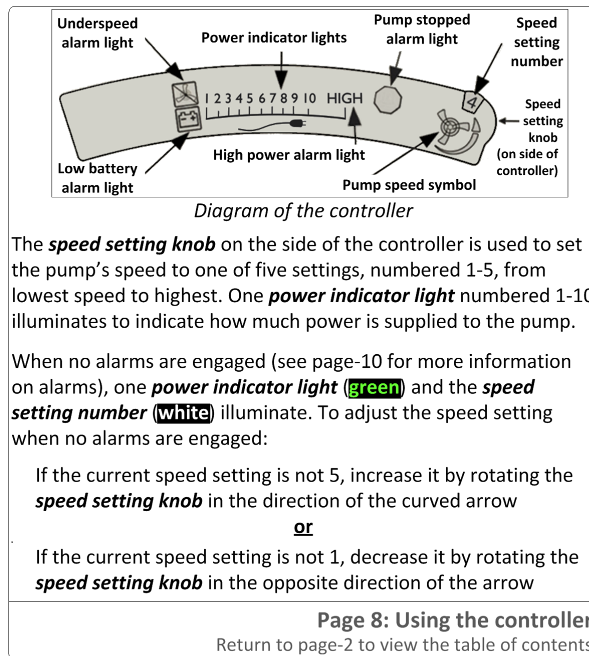
The draft manual is a graphically rendered prototype of a printed, spiral bound document based on the 29-page patient handbook from Chapter 5. To support the user in navigating through the

handbook, it has page-number labels in the bottom, left-hand corner of each page; cross-references; and a draft table of contents on page-2 (Fig. 9.4a). Content in the manual includes text and diagrams providing descriptions of the device, an operational procedure, and the pump stopped alarm troubleshooting procedure.

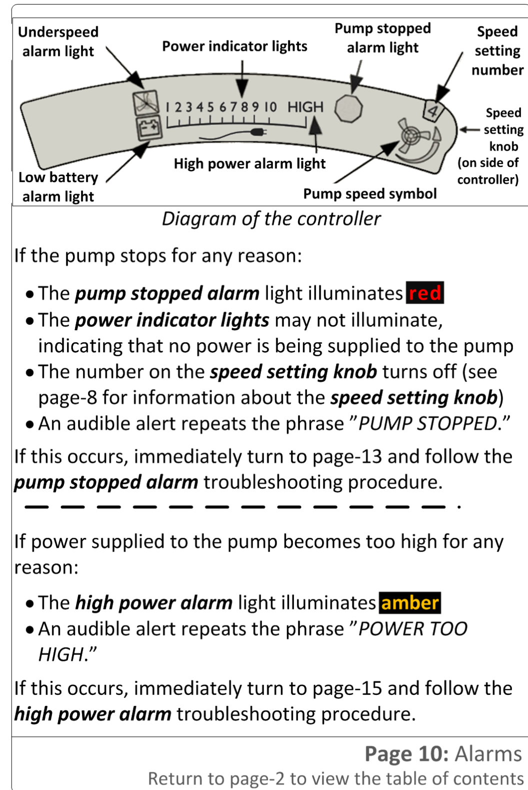
Using the controller 8
Alarms 10
Troubleshooting procedures
Pump stopped alarm 13
High power alarm 15

Page 2: Table of Contents

a



b



c

Figure 9.4: Graphical rendering of pages in the draft patient handbook

In this case study, the draft table of contents references four pages:

1. Page-8, which contains a labeled diagram of the controller, text explaining its functionality, and an operational procedure for adjusting pump speed (Fig. 9.3b)

2. Page-10, which contains a labeled diagram of the controller and information about the pump stopped and high power alarms (Fig. 9.3c)

3. Page-13, which contains the first four steps of the troubleshooting procedure for addressing the pump stopped alarm (continued onto page-14)

4. Page-15, which contains a troubleshooting procedure for addressing the high power alarm (procedure not developed or modeled in this case study)

Cross-references are added in Fig. 9.4 to aid the user in navigating between pages containing procedures and pages containing device descriptions. The current page number and a cross-reference to the table of contents are printed within the footer of every page in the body of the manual. Pages-13 and 15 (not shown in Fig. 9.4) also contain cross references back to page-10. This design was implemented because page-10 contains cross-references to pages 13 and 15, and the end user may need to navigate between them.

Text and labeled diagrams appear on pages-8 and 10 (Fig. 9.4b,c). Unlike the original handbook, the new design does not contain tables describing relationships between power indicator lights and pump speeds. This design was implemented to remove a potential usability problem uncovered in the model checking counterexample to *PumpSpeed visual documented redundancy* in Chapter 7, which revealed that power indicator lights on the controller and information within the patient handbook provide conflicting information about pump speed.

To support the end user in operating the device, the procedure on page-8 explains that the pump's speed setting can be adjusted by rotating the speed setting knob. To support the end user in troubleshooting the device, the draft manual contains a modified version of the pump stopped alarm troubleshooting instructions from Chapter 5. Setup, maintenance, and other troubleshooting procedures (e.g. high power alarm) are not considered in this case study.

The modified pump stopped alarm troubleshooting procedure has six main steps with sub-steps for completing tasks that are applicable to different initial hardware configurations. Its contents are similar to the original procedure with respect to what the user must do, such as attempting to fix a broken connector. Leveraging design principles from Chapter 1, the modifications listed below aim to improve usability of the procedure with respect to completeness, accuracy, and time efficiency:

1. Task descriptions were designed for completeness such that the procedure is applicable to all initial hardware configurations

2. Device descriptions were designed for accuracy such that tasks involving multi-part components

identify the part on which the user should act

3. The procedure was designed for time efficiency such that:

   (a) Actions that could correct the problem come before actions that cannot

   (b) Text that was deemed unnecessary or potentially problematic in Chapter 5 was removed

In support of completeness, separate sets of sub-steps are developed to address initial component configurations that are possible when the pump stopped alarm engages. Modifications to the original procedure employing this principle appear in steps 1–4 of the modified procedure.

In support of accuracy, input/output ends of cables are identified specifically within steps of the procedure, and a diagram similar to Fig. 5.3 from Chapter 5 appears pages containing steps of the procedure. In addition to describing parts of components explicitly, the procedure includes references to the diagram having graphical renderings of components and captions describing them (Fig. 9.2). Modifications to the original procedure employing this principle appear in all steps of the modified procedure.

In support of time efficiency, steps leveraged from the original procedure are reordered to prioritize steps that could potentially resolve the alarm over ones that cannot. A new version of steps 4a–b from the original procedure (fixing the connector permanently attached to the heart) is placed at the beginning of the modified procedure. The original step-3 (attached a red tag to old components) is moved to the end (step-6 of the modified procedure), as it cannot resolve the alarm. The original steps- 2b and 5a are switched so the end user attempts connecting a lead reserve battery to the new controller before connecting a new lithium-ion battery. This modification was implemented to reduce the number of actions: connecting the lead battery requires one action that could potentially restart the pump, while connecting a lithium-ion battery requires two actions.

In support of time efficiency, text within the original procedure that was considered unnecessary or potentially problematic was removed from the modified procedure. Step 1a of the original procedure had text instructing the end user to disconnect the lithium-ion battery cable. The formal task model representation of this step enabled a potentially unsafe situation to emerge later in the

procedure (reconnecting a potentially discharged or malfunctioning battery). Thus, it was removed from the modified procedure. Step 6b of the original procedure had text explaining why a power source should be disconnected from the controller if prior steps were unsuccessful for resolving the alarm. As mentioned in Chapter 5, this text was not represented within the formal task model, as it was not necessary for understanding what components are involved in the task and how it should be executed. Thus, the modified procedure does not include this text. Instead, text is added in Step 5(c)i to provide declarative knowledge regarding what is displayed on the controller when the task has been completed: no power indicator lights are illuminated on the controller.

The modified procedure is shown in the grey box below exactly as it is written on pages 13–14 of the prototype patient handbook. Steps 1–4 appear on page-13, while steps-5 and 6 continue onto page-14.

Follow this procedure if the ***pump stopped alarm*** is on (see page-10 for more information on the ***pump stopped alarm***). To complete the following steps safely use the bold letters in parentheses referencing components shown in **Box 1**.

1. Check to see if the connector permanently attached to the heart **(f)** is broken. If it is not broken, proceed immediately to step-2. If it is broken:
   (a) Put it back together where it is broken. If the ***pump stopped alarm*** turns off, discontinue this procedure. Otherwise:
      i. Take the connector apart again, rotate the parts 90 degrees, and put it back together again
      ii. If the ***pump stopped alarm*** is still on, repeat steps 1a-i three times
      iii. If the ***pump stopped alarm*** turns off, discontinue this procedure
2. If the abdominal cable is in-use:
   (a) Disconnect output **(h)** of the pump cable from input **(i)** of the abdominal cable
   (b) Loosen the 1/2 cell battery cap on the controller **(b)** to silence the audible alarm
   (c) Leave all cables attached to the controller and battery and set them aside

   Otherwise, if the abdominal cable is not in-use:
   (d) Disconnect output **(g)** of the pump cable from input **(a)** of the controller
   (e) Loosen the 1/2 cell battery cap on the controller **(b)** to silence the audible alarm
   (f) Leave all cables attached to the controller and battery and set them aside
3. Retrieve a replacement controller and:
   (a) Connect output **(g)** of the pump cable to input **(a)** of the replacement controller
   (b) Tighten the 1/2 cell battery cap **(b)** to activate the audible alarm
4. If you have the lead battery with you (cable segment shown in **Box 1, (m)**) and it was not connected when the ***pump stopped alarm*** turned on:
   (a) Connect output **(m)** of the lead battery cable to input **(c)** of the replacement controller
   (b) If the pump stopped alarm turns off, proceed to step-6. If the ***pump stopped alarm*** remains on, disconnect output **(m)** of the lead battery cable from input **(c)** of the replacement controller

Otherwise, if you do not have the lead battery with you or you have already completed steps 4a–b, retrieve a replacement lithium-ion battery and proceed to step-5

5. While depressing the black button **(o)** on a replacement lithium-ion battery, make sure five charge level indicator lights **(q)** illuminate. If four or fewer lights illuminate, call your emergency number immediately and have the patient lie down in a comfortable position while awaiting emergency help

   Otherwise, if five lights illuminate:

   (a) Retrieve a replacement lithium-ion battery cable and connect output **(e)** to input **(c)** of the replacement controller
   (b) Connect output **(d)** of the replacement lithium-ion battery cable to input **(q)** of the replacement lithium-ion battery
   (c) If the ***pump stopped alarm*** remains on:
      i. Disconnect output **(e)** of the replacement lithium-ion battery cable from input **(c)** of the replacement controller, ensuring that no power indicator lights on the controller are illuminated green
      ii. Call your emergency number immediately
      iii. Have the patient lie down in a comfortable position while you await emergency help

6. Attach a red tag **(r)** to the old components you set aside earlier

Figure 9.5: Outline form of case study troubleshooting procedure

9.4 Integrated Framework Model

To analyze the case study system, the integrated framework architecture was implemented by encoding and composing:

- A *documentation navigation* model representing the end user navigating through the patient handbook using its navigational tools
- Two *task* models representing the end user executing:
  - The draft operational procedure
  - The draft pump stopped alarm troubleshooting procedure
- A *device* model, including:
  - A *display/control logic* model representing the system's internal algorithms and what is displayed on the device
  - An *end user-device interaction* model representing changes to controls and configurable hardware that emerge following end-user inputs

  - A *plant* model representing pump speed and power supplied by the controller
  - A *constraints* model that aids in constraining model checking analyses to *plant* model outputs matching simulation data

- An *affordance* model representing opportunities for the end user to execute actions needed to resolve the pump stopped alarm
- A *signifier* model representing what is signified to the end user through audible, visual, and documentation channels

In regard to locating pages of the patient handbook having necessary content, the end user should be able to navigate to pages of the handbook containing procedural steps and text, tables, and diagrams explaining what is signified through audible and visual channels. To evaluate usability of the interface for an end user having no training or experience using the device, the model precludes any possibility of the end user remembering documented procedural steps or device information. Thus, the *documentation navigation* model page number output operates as an input to *task* and *signifier* models. Preconditions of formal task model activities specify that procedural steps listed on that page could begin executing when the end user is on the page. Otherwise, if a page containing procedural steps cannot be locating using the patient handbook's navigational tools, those steps cannot execute. In the *signifier* model, the page number output operates as an input such that a function or meaning is signified through the documentation channel only if the explanation is provided on the current page. Otherwise, nothing is signified through the documentation channel.

In regard to understanding of what is signified, signifiers on the device should be consistent, redundant (if applicable, e.g. when the audible/visual pump stopped alarm is engaged), and updated in a time-efficient way when operational states of the device change. To enable analyses of these characteristics, input/output variables are exchanged among the *task*, *end user-device interaction*, *display/control logic*, *plant*, and *signifier* models. Changes to controls and configurable hardware are represented via human action outputs of the *task* models operating as inputs to the *end user-device interaction* model (as in Chapter 5).

9.5 Documentation Navigation

The *documentation navigation* model represents the end user navigating through the 29-page patient handbook using its navigational tools, including sequentially ordered, numerically labeled pages; a main table of contents on page-2; text, tables, and diagrams on pages 8, 10, 13, 14, and 15; and cross-references on pages 2, 8, 10, 13, 14, and 15. Page-2 is modeled as the initial page. Transitions represent the end user remaining on pages having text, tables, or diagrams (8, 10, 13, 14, and 15); turning from page 13 to 14 to continue the pump stopped alarm troubleshooting procedure; and navigating to any cross-referenced page in one step. Cross-referenced pages are listed below. Each page having cross-references is listed as "Page number. *Page title*," where the page number and title appear in the bottom left-hand corner (Fig. 9.4).

2. *Table of contents*: 8, 10, 13, 15
8. *The controller*: 2, 10
10. *Alarms*: 2, 8, 13, 15
13. *Pump stopped alarm*: 2, 10
14. *Pump stopped alarm*: 2
15. *High power alarm*: 2, 10

The model was 27 lines of SAL code (Appendix H.2.1).

9.6 Task Models

Applying the formalism/tool described in [10] and the modeling technique discussed in Section 9.1, two task models are encoded to represent procedures in the prototype patient handbook. The first model represents the procedure for adjusting pump speed, which is listed in text on page-8 (Fig. 9.4c). The second model represents the troubleshooting procedure for addressing the pump stopped alarm: steps 1–4 are listed on page-13, and steps 5–6 are listed on page-14.

The formal task models are visualized using the graphical notation of EOFM [73]. Activities not at the lowest level begin with *a* for activity and are surrounded by rectangles with rounded edges. The

lowest level activities in the tree begin with *h* for human action and are represented inside rectangles with right-angle edges. Preconditions (conditions for when an activity can initiate) in the figures are denoted by yellow triangles pointing downward. Completion conditions (conditions for when an activity has been completed) are denoted by pink triangles pointing upward. Input variables, which begin with a lowercase *i*, are used in preconditions, repeat conditions, and completion conditions to govern activity/action execution. The final EOFM-XML model representing both procedures was 253 lines of XML code (Appendix H.1.1). The automated translator described in [10] generated a 754-line SAL model (Appendix).

9.6.1 Pump Speed Adjustment

The end user could either increase or decrease the speed setting if the current setting is not desired. As indicated by the procedure, it is only possible to increase the setting if it is not already at the maximum setting of 5, and it is only possible to decrease if it is not already at the minimum setting of 1. Additionally, the manual indicates that speed setting should be adjusted only when no alarms are engaged. This information is utilized within activity execution conditions.

In the formal task model (Fig. 9.6), there is one top-level activity named *aAdjustPumpSpeed* and two sub-activities named *aIncreaseSpeed* and *aDecreaseSpeed* respectively. The pump's speed setting is an input variable with prefix *i*), *iSpeedSetting*, which could be a numeric value between 1 and 5. The user's desired speed is a local variable (with prefix *l*), *lDesiredSetting*. Its initial value is assigned randomly. A second input variable *iAlarm* is incorporated within the precondition to specify that no alarms should be engaged. The completion condition specifies that the procedure has finished executing when the speed setting is equal to the end user's desired speed.

Text in the procedure identifies two different sets of preconditions for increasing or deceasing speed as well as what component should be acted on (the speed setting knob); thus, this activity is decomposed by *xor* into two sub-activities:

1. The first sub-activity (*aIncreaseSetting*) represents the task of increasing the speed setting by rotating the knob in the direction of the curved arrow (counterclockwise). The precondition specifies that the activity begins executing if the speed setting is less than the desired speed

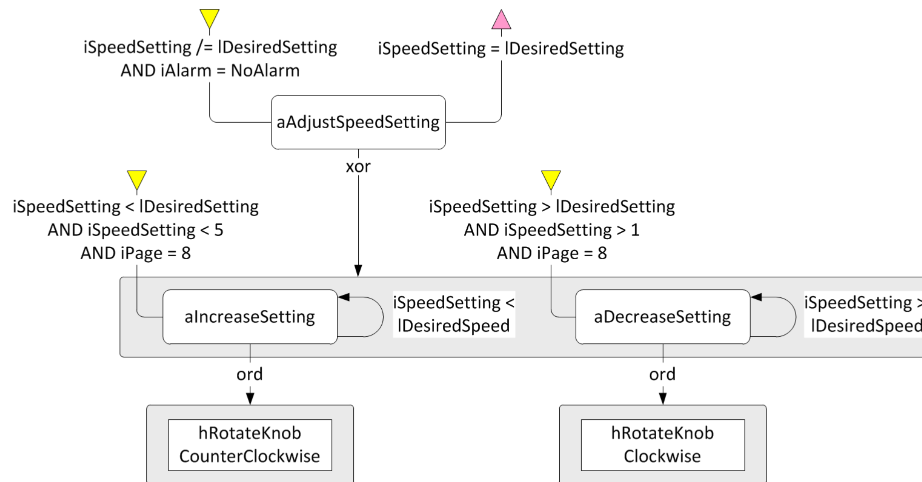


Figure 9.6: Graphical representation of the task model representing the user manual procedure for adjusting speed setting on the pump's controller

(*iSpeedSetting* < *lDesiredSpeed*); the speed setting is below the maximum (*iSpeedSetting* < *5*); and, because text prescribing this text is listed on page-8 of the user manual, the end user is on page-8 (*iPage* = 8). The repeat condition specifies that the activity repeats execution if the speed setting is less than the desired speed. The activity is decomposed by *ord* into one human action representing the end user rotating the speed setting knob counterclockwise (*hRotateKnobCounterClockwise*)

2. The second sub-activity (*aDecreaseSetting*) represents task of decreasing the speed setting by rotating the knob in the opposite direction of the curved arrow (clockwise). The precondition specifies that the activity begins executing if the speed setting is greater than the desired speed (*iSpeedSetting* > *lDesiredSpeed*); the speed is setting is above the minimum (*iSpeedSetting* > *1*); and, because text prescribing this text is listed on page-8 of the user manual, the end user is on page-8 (*iPage* = 8). The repeat condition specifies that the activity repeats execution if the speed setting is greater than the desired speed. The activity is decomposed by *ord* into one human action representing the end user rotating the speed setting knob clockwise (*hRotateKnobClockwise*)

9.6.2 Pump Stopped Alarm Troubleshooting

The six main steps of the modified troubleshooting procedure are depicted as six sub-activities of a top-level activity *aRespondToPumpStoppedAlarm*, representing the entire procedure (Fig. 9.7). Individual steps and sub-steps are depicted in Figs. 9.8–9.13. The top-level activity in Fig. 9.7 has execution conditions specifying that the procedure begins executing when the pump stopped alarm is engaged and completes execution when the alarm disengages (modeling of the alarm status discussed later).

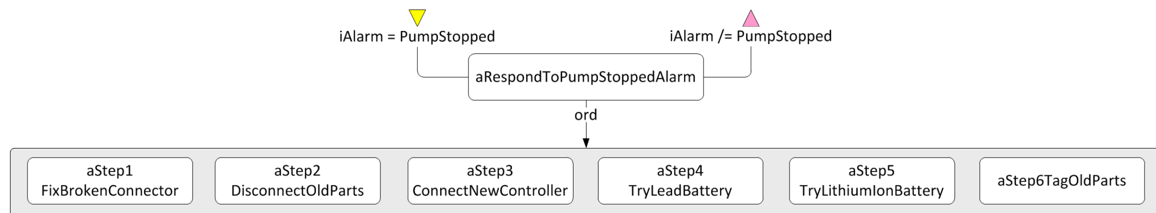


Figure 9.7: Visualization of the six main steps of the modified troubleshooting procedure encoded as six EOFM sub-activities. The top-level activity *aRespondToPumpStoppedAlarm* represents the entire procedure, while the six sub-activities represent end-user activities prescribed within the six main steps. A top-level activity precondition specifies that the procedure begins executing when the pump stopped alarm engages. A completion condition specifies that the procedure completes execution when the alarm disengages. The *ord* decomposition operator specifies that all six sub-activities must execute in order

Step 1 (*aStep1FixBrokenConnector*) appears in Fig. 9.8. The precondition specifies that the activity begins executing if the connector permanently attached to the heart is broken (*iPermanentlyAttachedConnector = Broken*). Because text instructing this step is on page-13 of the patient handbook, the precondition also specifies that the end user is on page-13 (*iPage = 13*). The activity is decomposed by *ord* into two sub-activities representing sub-tasks of reassembling the broken connector and rotating connector parts. Text of the modified procedure instructing these sub-tasks is similar to step 4 of the original procedure. The sub-activities representing them are the same as *aReassembleConnector* and *aTryRotatingParts* of Chapter 5.

Step 2 (*aStep2DisconnectOldParts*) appears in Fig. 9.9. Because text instructing this step is on page-13 of the patient handbook, the precondition specifies that the activity begins executing if the end user is on page-13 (*iPage = 13*). The activity is decomposed by *ord* into three sub-activities having different preconditions:

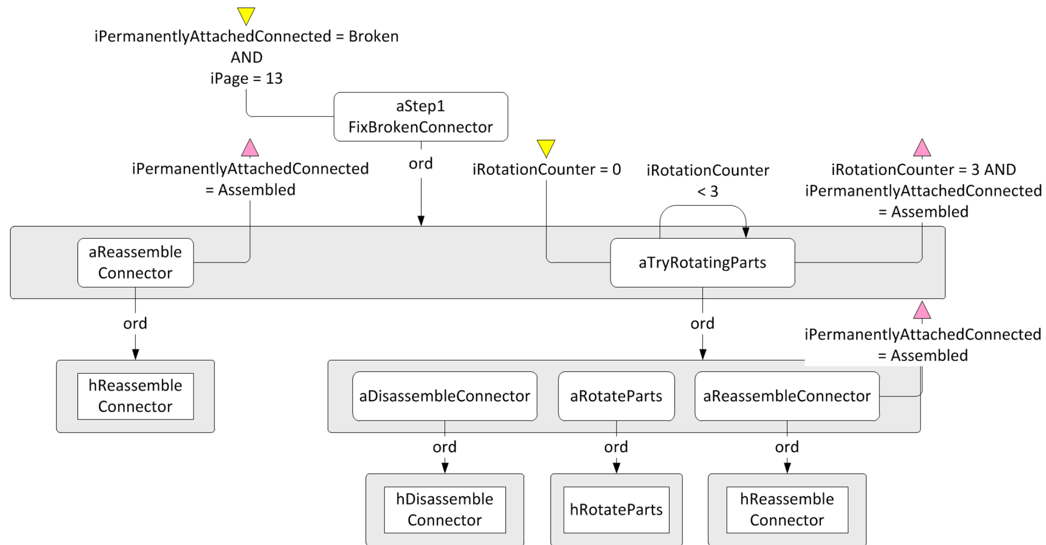


Figure 9.8: Visualization of the formal task model representing step-1

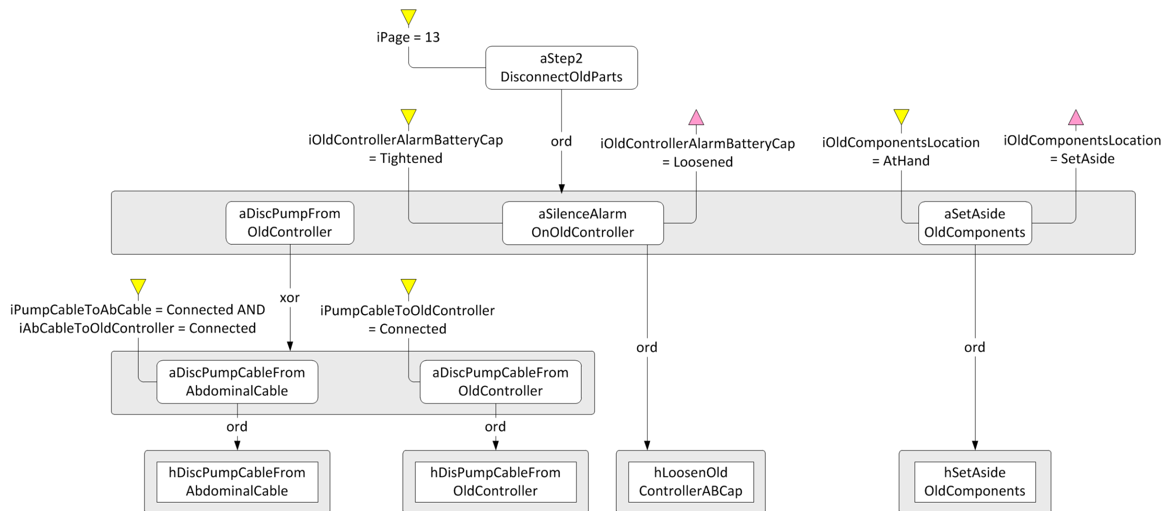


Figure 9.9: Visualization of the formal task model representing step-2

1. The first sub-activity (*aDiscPumpFromOldController*) represents the first sub-task in the text, which instructs the end user to disconnect the pump cable from either the old controller or the abdominal cable. Because text identifies two sets of sub-tasks that address different component configurations, the activity is decomposed by *xor* into two sub-activities:

   (a) The first sub-activity (*aDisPumpCableFromAbdominalCable*) represents the sub-task that is possible if the abdominal cable is in-use. The precondition specifies that the activity begins executing if the pump cable is connected to the abdominal cable (*iPumpCable-*

ToAbCable = Connected) and the abdominal cable is connected to the old controller (*iAbCableToOldController = Connected*). Because text identifies what parts should be acted on (the pump cable output end and the abdominal cable input socket), the activity is decomposed by *ord* into one human action representing the end user disconnecting the pump cable from the abdominal cable (*hDiscPumpCableFromAbdominalCable*)

(b) The second sub-activity (*aDiscPumpCableFromOldController*) represents the sub-task that is possible if the abdominal cable is not in-use. The precondition specifies that the activity begins executing if the pump cable is connected to the old controller (*iPumpCableToOldController = Connected*). Because text identifies what parts should be acted on (the pump cable output end and the old controller's pump cable input socket), the activity is decomposed by *ord* into one human action representing the end user disconnecting the pump cable from the old controller (*hDiscPumpCableFromOldController*)

2. The second sub-activity (*aSilenceAlarmOnOldController*) represents the second sub-task, which instructs the end user to silence the alarm by loosening the 1/2 cell alarm battery cap on the old controller. Text of the modified procedure instructing this sub-task is similar to the last sub-task in step 1a of the original procedure. The sub-activity representing it is the same as *aSilenceAlarmOnOldController* of Chapter 5.

3. The third sub-activity (*aSetAsideOldComponents*) represents the third sub-task, which instructs the end user to set aside old components. Text of the modified procedure instructing this task is similar to the original procedure. The sub-activity representing it is the same as *aSetAsideOldComponents* of Chapter 5.

Step 3 (*aStep3ConnectNewController*) appears in Fig. 9.10. Because text instructing this step is on page-13 of the patient handbook, the precondition specifies that the activity begins executing if the end user is on page-13 (*iPage = 13*). Text of the modified procedure instructing the two sub-tasks is similar to the original procedure. The sub-activities representing them are the same as *aConPumpCableTpNewController* and *aActivateAlarmOnNewController* of Chapter 5.

Step 4 (*aStep4TryLeadBattery*) appears in Fig. 9.11. Because text instructing this step is on page-13 of the patient handbook, the precondition specifies that the activity begins executing if the end user is on page-13 (*iPage = 13*). The precondition also specifies that the activity begins executing if the lead battery was not in-use when the pump stopped alarm engaged (*iLeadBattToOldController = Disconnected AND NOT(iLeadBattToYCable = Connected AND iYCableToOldController = Connected)*). The activity is decomposed by *ord* into two sub-activities representing two subtasks prescribed in the text:

1. The first sub-activity (*aConLeadBattToNewController*) represents the first sub-task, which instructs the end user to connect the lead battery to the new controller. The precondition specifies that the activity begins executing if the lead battery is disconnected from the new controller (*iLeadBattToNewController = Disconnected*). The completion condition specifies that the activity has completed executing when the lead battery is connected to the new controller (*iLeadBattToNewController = Connected*). Because text identifies what part should be acted on (the lead battery cable output end), the activity is decomposed by *ord* into one human action representing the end user connecting the lead battery to the new controller (*hConLeadBattToNewController*)

2. The second sub-activity (*aDiscLeadBattFromNewController*) represents the second sub-task

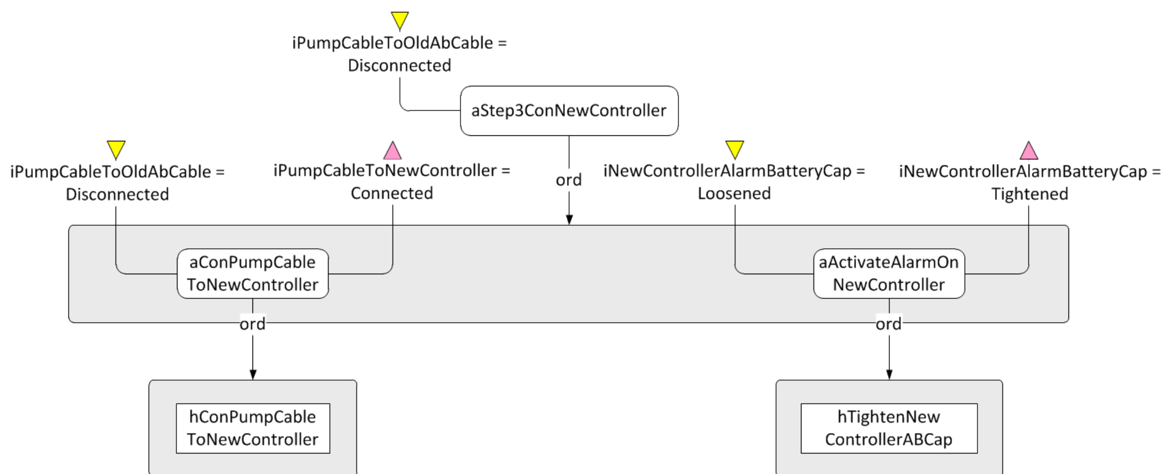


Figure 9.10: Visualization of the formal task model representing step-3

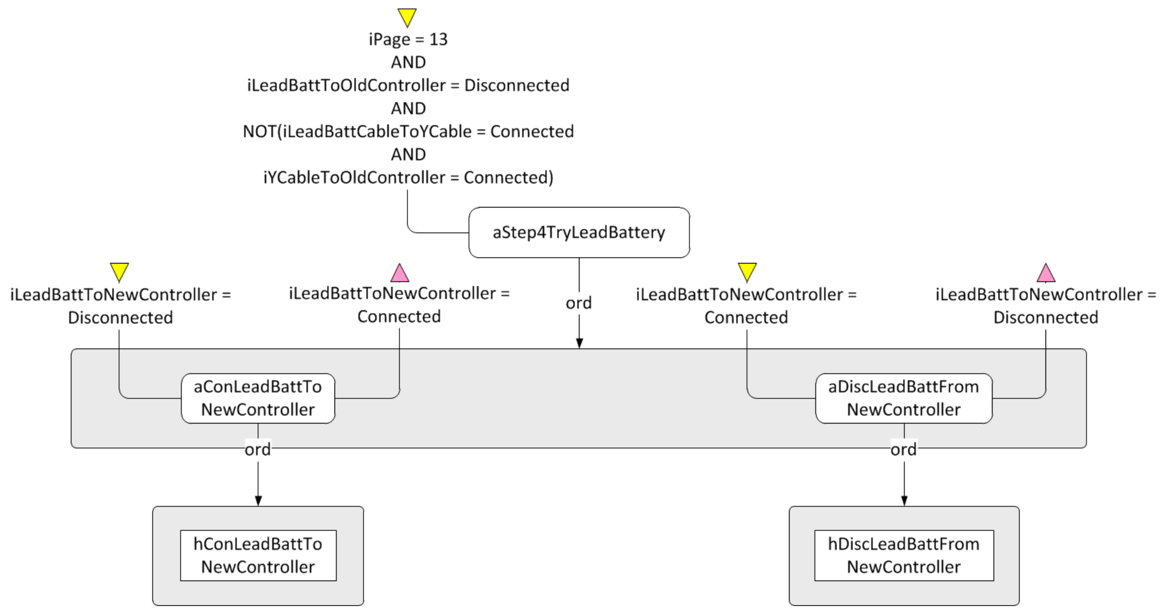


Figure 9.11: Visualization of the formal task model representing step-4. The activities *aCallEmergencyNumber* and *aCallEmergency* represent the same task. This task appears twice because the first instance, *aCallEmergencyNumber*, has a precondition (*iNewLiBattLights* $<$ *5*) while the second instance, *aCallEmergency*, has no precondition

prescribed in the text, which instructs the end user to disconnect the lead battery from the new controller if connecting it did not resolve the alarm (discussed further in Section 9.8.1). The precondition specifies that the activity begins executing if the lead battery is connected to the new controller (*iLeadBattToNewController* = *Connected*). The completion condition specifies that the activity has completed executing when the lead battery is disconnected from the new controller (*iLeadBattToNewController* = *Disconnected*). It is decomposed by *ord* into one human action representing the end user disconnecting the lead battery from the new controller (*hDiscLeadBattFromNewController*)

Step 5 (*aStep5TryLiIonBattery*) appears in Fig. 9.12. Because text instructing this step is on page-14 of the patient handbook, the precondition specifies that the activity begins executing if the end user is on page-14 (*iPage* = *14*). The activity is decomposed into two sub-activities representing the two sub-tasks prescribed in text:

1. The first sub-activity (*aCheckNewLiBattLevel*) represents the first sub-task, which instructs the end user to check the new lithium-ion battery's charge level. Because text identifies what

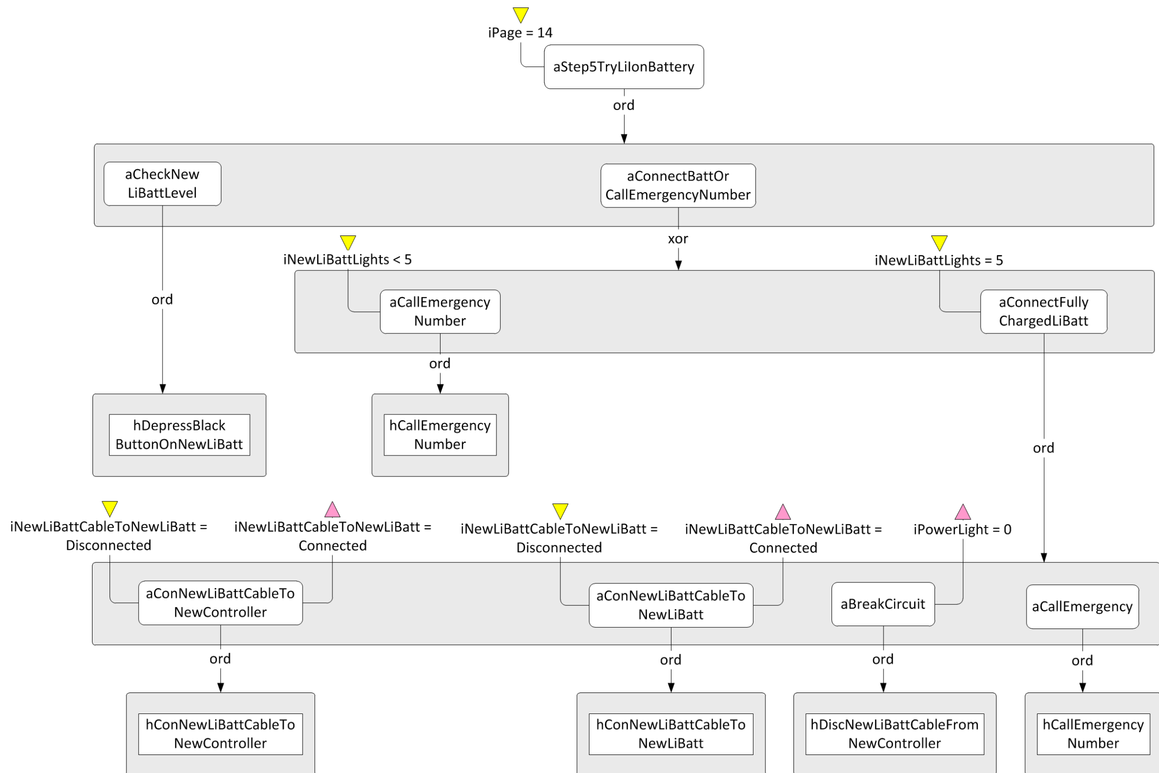


Figure 9.12: Visualization of the formal task model representing step-5

part should be acted on (black button on the battery), the activity is decomposed by *ord* into one human action representing the end user depressing the black button to check the battery's charge level (*hDepressBlackButtonOnNewLiBatt*)

2. The second sub-activity (*aConnectBattOrCallEmergencyNumber*) represents the second sub-task. It is decomposed by *xor* into two sub-activities, one of which could execute under different preconditions:

   (a) The first sub-activity (*aCallEmergencyNumber*) represents the sub-task that could execute if the new lithium-ion battery is not fully charged. The precondition specifies that the activity begins executing if fewer than five lights illuminate on the new lithium-ion battery ($iNewLiBattLights < 5$). It is decomposed by *ord* into one human action representing the end user calling an emergency number (*hCallEmergencyNumber*)

   (b) The second sub-activity (*aConnectFullyChargedLiBatt*) represents the other sub-task that

could execute if the new lithium-ion battery is fully charged. The precondition specifies that the activity begins executing if all five lights illuminate on the new lithium-ion battery (*iNewLiBattLights* = *5*). It is decomposed by *ord* representing the three sub-tasks prescribed in text:

i. The first sub-activity (*aConNewLiBattCableToNewController*) represents the first sub-task, which instructs the end user to connect the new lithium-ion battery battery to the new controller. The precondition specifies that the activity begins executing if the new lithium-ion battery cable is disconnected from the new controller (*iNewLiBattCableToNewController* = *Disconnected*). The completion condition specifies that the activity has completed executing when the cable is connected (*iNewLiBattCableToNewController* = *Connected*). It is decomposed by *ord* into one human action representing the end user connecting the lead battery to the new controller (*hConLeadBattToNewController*)

ii. The second sub-activity (*aConNewLiBattCableToNewLiBatt*) represents the second sub-task prescribed in the text, which instructs the end user to connect the new lithium-ion battery cable to a fully charged lithium-ion battery. The precondition specifies that the activity begins executing if the new lithium-ion battery cable is disconnected from the new lithium-ion battery (*iNewLiBattCableToNewLiBatt* = *Disconnected*). The completion condition specifies that the activity has completed executing when the cable has been connected (*iNewLiBattCableToNewLiBatt* = *Connected*). Because text identifies what parts should be acted on (a new lithium-ion battery cable output end and the new lithium-ion battery input socket), the activity is decomposed by *ord* into one human action representing the end user connecting the new lithium-ion battery cable to the new lithium-ion battery (*hConNewLiBattCableToNewLiBatt*)

iii. The third sub-activity (*aBreakCircuit*) represents the third sub-task, which instructs the end user to break the circuit if connecting the new lithium-ion battery did not

resolve the alarm (discussed further in Section 9.8.1). The completion condition specifies that the activity has completed executing when no power lights are illuminated on the controller (*iPowerLight = 0*). Because text identifies what part should be acted on (the lithium-ion battery cable output end connected to the new controller), the activity is decomposed by *ord* into one human action representing the end user disconnecting the lithium-ion battery cable from the new controller (*hDiscNewLiBattCableFromNewController*)

iv. The fourth sub-activity (*aCallEmergency*) represents the last sub-task, which instructs the end user to call an emergency number. It is decomposed by *ord* into one human action representing the end user calling an emergency number (*hCallEmergencyNumber*)

Step 6 (*aStep6TagOldParts*) appears in Fig. 9.13, which instructs the end user to attach a red tag to the old components set aside in step 2c. Because text instructing this step is on page-14 of the patient handbook, the precondition specifies that the activity begins executing if the end user is on page-14 (*iPage = 14*). Text of the modified procedure instructing this task is similar to the original procedure. The sub-activity representing it is the same as *aStep3RedTagOldComponents* of Chapter 5.

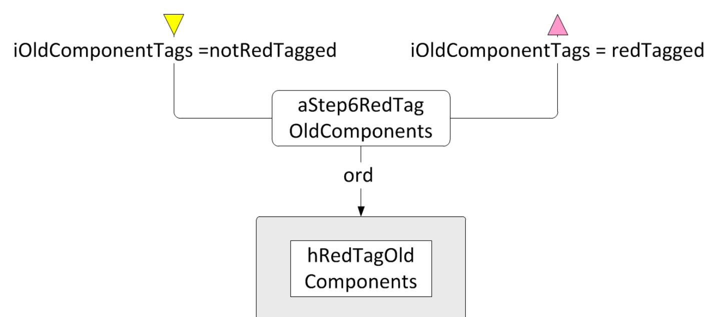


Figure 9.13: Visualization of the formal task model representing step-6

9.7 Affordances

Using the CAVEMEN approach (Chapter 6), an *affordance* model was encoded to represent configurable hardware components shown in Fig. 9.2a–o and thirteen affordances involving them. 11 of

13 correspond to human action variables from the pump stopped alarm troubleshooting procedure *task* model. All 11 involve connecting cables, disconnecting cables, or repairing the connector permanently attached to the heart. The purpose of encoding these 11 affordances was to demonstrate an implementation of the framework integrating *task*, *affordance*, and *end user-device interaction* models. The 11 EOFM human action variables and corresponding affordances are listed below as "*action variable–affordance variable*":

1. *hReassembleBrokenConnector–ConnectorPartsAssemblable*
2. *hDisassembleConnector–ConnectorPartsDisassemblable*
3. *hRotateConnectorParts–ConnectorPartsRotatable*
4. *hDiscPumpCableFromAbCable–PumpCableDisconnectableFromAbCable*
5. *hDiscPumpCableFromOldController–PumpCableDisconnectableFromOldController*
6. *hConPumpCableToNewController–PumpCableConnectableToNewController*
7. *hConLeadBattToNewController–LeadBattConnectableToNewController*
8. *hDiscLeadBattFromNewController–LeadBattDisconnectableFromNewController*
9. *hConNewLiBattCableToNewController–NewLiBattCableConnectableToNewController*
10. *hConNewLiBattCableToNewLiBatt–NewLiBattCableConnectableToNewLiBatt*
11. *hDiscNewLiBattCableFromNewController–NewLiBattCableDisconnectableFromNewController*

Two of 13 affordances enable the end user to connect the old lithium-ion battery to the new controller during the pump stopped alarm troubleshooting procedure, and corresponding actions are not represented in the *task* model. The purpose of encoding these affordances was to enable formal verification of usability specifications involving error tolerance (discussed later). One affordance variable (*NewLiBattCableConnectableToOldLiBatt*) represents the end user's opportunity to connect the new lithium-ion battery cable to the old lithium-ion battery. The other variable (*OldLiBattCableConnectableToNewController*) represents the end user's opportunity to connect the old

lithium-ion battery cable to the new controller. Both affordances are considered unsafe if the old lithium-ion battery was in-use when the pump stopped alarm engaged.

The 398-line CAVEMEN-XML model (described in Section 9.7.1) was translated to 290 lines of SAL code using the JavaScript-based translator described in Chapter 6. The automatically generated HES model was augmented with manually encoded infrastructure representing static end-user motor capabilities and evolving spatial relations among HES entities (described in Section 9.7.2). The final model was 486 lines of SAL code (Appendix H.2.3).

9.7.1 CAVEMEN-XML Model

10 configurable hardware components and 13 affordances were encoded using CAVEMEN-XML. Configurable hardware specifications are described in Section‘9.7.1.1. Affordance specifications are described in Sections 9.7.1.3–9.7.1.5. To demonstrate an application of CAVEMEN-XML to specify Chemero's formalism, all 13 *affordance* nodes have the *formalism* attribute valued *chemero*.

9.7.1.1 Modelobject, Subobject, and Atomicobject Nodes

10 configurable hardware components are involved in the 13 affordances: two controllers, two lithium-ion battery cables, one pump cable, one abdominal cable, one Y-Cable, two lithium-ion batteries, and one lead battery. Components were encoded within ten CAVEMEN-XML *modelobject* nodes using 42 lines of CAVEMEN-XML (Appendix H.1.2, lines 4–46). For each *modelobject* node, parts of components involved in the affordances were represented within *subobject* and *atomicobject* nodes. Parts that are not involved in these affordances, such as the 1/2 cell AA battery caps of controllers, were not specified.

For the two controllers, parts considered in the case study include one input socket for the pump cable/abdominal cable and one input socket for a battery cable (lead, lithium-ion, or Y-cable). CAVEMEN-XML nodes representing the old controller and its two input sockets are shown in Fig. 9.14a–c. The controller is specified using a *modelobject* node with the *name* attribute valued *mOldController* (Fig. 9.14a). The input socket for the pump cable of abdominal cable output end is specified using an *atomicobject* node with the *name*attribute *aoOCPumpInput*, where "OC" stands

for "old controller" (Fig. 9.14b). The input socket for the lead, Y, or lithium-ion battery cable is specified using an *atomicobject* node with the *name*attribute *aoOCBatteryInput* (Fig. 9.14c). The new controller is encoded in a similar way, replacing the *modelobject name*attribute value with *mNewController* and each instance of "OC" with "NC," where "NC" stands for "new controller."

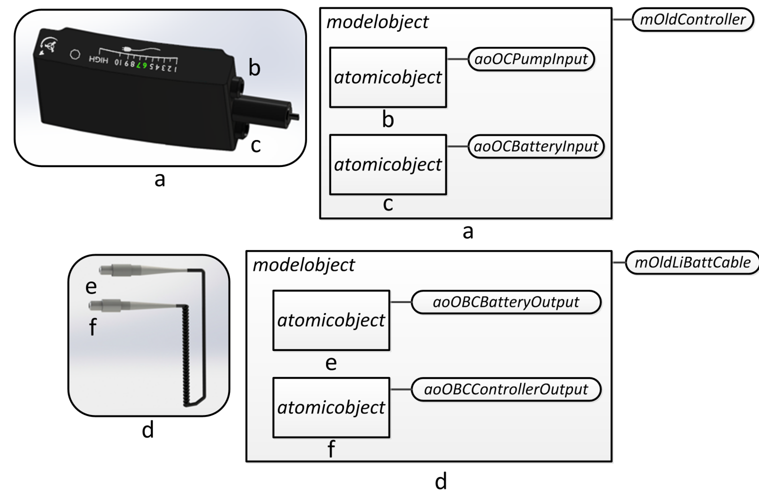


Figure 9.14: Graphical representation of the case study system's old controller, old lithium-ion battery cable, and CAVEMEN-XML nodes representing them. Node names are listed within square-edge rectangles. *Name* attribute values are listed within rounded-edge rectangles. Letters aid in identifying what components correspond to *modelobject* nodes and what parts correspond to *subobject* and *atomicobject* nodes

For the two lithium-ion battery cables, parts include controller and battery output ends. CAVEMEN-XML nodes representing the old lithium-ion battery and its two output ends are shown in Fig. 9.14d–f. The cable is specified using a *modelobject* node with the *name*attribute valued *mOldLiBattCable* (Fig. 9.14d). The output end connecting to a lithium-ion battery is specified using an *atomicobject* node with the *name*attribute *aoOBCBatteryOutput*, where "OBC" stands for "old lithium-ion battery cable" (Fig. 9.14e). The output end connecting to a controller or Y-Cable input socket is specified using an *atomicobject* node with the *name*attribute *aoOBCControllerOutput* (Fig. 9.14f). The new lithium-ion battery cable is encoded in a similar way, replacing the *modelobject name*attribute value with *mNewLiBattCable* and each instance of "OBC" with "NBC," where "NBC" stands for "new lithium-ion battery cable."

For the pump cable, parts include the output end connecting to a controller or abdominal cable input socket, the connector permanently attached to the heart, and the two connector parts.

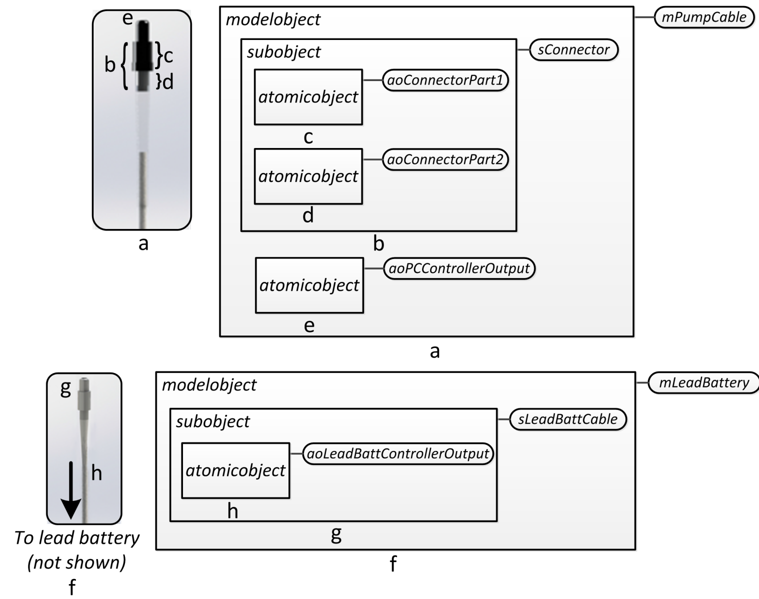


Figure 9.15: Graphical representation of the case study system's pump cable, lead battery, and CAVEMEN-XML nodes representing them. Node names are listed within square-edge rectangles. *Name* attribute values are listed within rounded-edge rectangles. Letters aid in identifying what components correspond to *modelobject* nodes and what parts correspond to *subobject* and *atomicobject* nodes

CAVEMEN-XML nodes representing the pump cable are shown in Fig. 9.15a–e. The cable is specified using a *modelobject* node with the *name*attribute valued *mPumpCable* (Fig. 9.15a). The connector permanently attached to the heart is specified using a *subobject* node with the *name*attribute valued *sConnector* (Fig. 9.14b). The two connector parts are specified using *atomicobject* nodes with *name*attributes valued *aoConnectorPart1* and *aoConnectorPart2* respectively (Fig. 9.14c, d). The output end connecting to a controller or abdominal cable input socket is specified using an *atomicobject* node with the *name*attribute *aoPCControllerOutput*, where "PC" stands for "pump cable" (Fig. 9.14e).

For the lead battery, parts include the cable and its output end connecting to a controller or Y-cable input socket. CAVEMEN-XML nodes representing the lead battery are shown in Fig. 9.14f–h. The battery is specified using a *modelobject* node with the *name*attribute valued *mLeadBattery* (node shown in Fig. 9.14f, battery not shown). The cable is specified within a *subobject* node with the *name*attribute valued *sLeadBattCable* (Fig. 9.14g). The output end connecting to a controller

or Y-Cable is specified using an *atomicobject* node with the *name*attribute valued *aoLeadBattControllerOutput*(Fig. 9.14h).

For the abdominal cable, parts include the pump input socket and controller output end. CAVEMEN-XML nodes representing the abdominal cable are shown in Fig. 9.16a–c. The cable is specified using a *modelobject* node with the *name*attribute valued *mAbdominalCable* (Fig. 9.16a). The input socket for the pump cable is specified using an *atomicobject* node with the *name*attribute valued *aoACPumpInput*, where “AC” stands for “abdominal cable” (Fig. 9.16b). The output end connecting to a controller input socket is specified using an *atomicobject* node with the *name*attribute *aoACControllerOutput* (Fig. 9.16c).

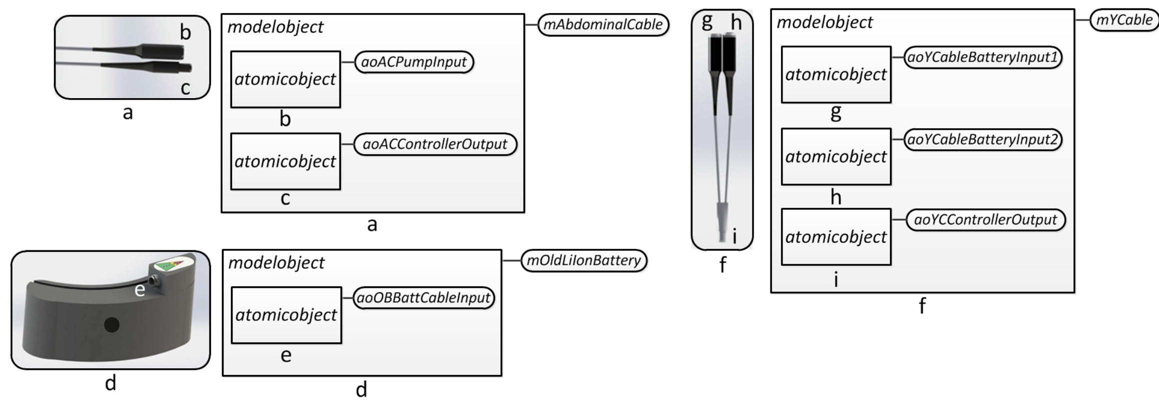


Figure 9.16: Graphical representation of the case study system’s abdominal cable, old lithium-ion battery, Y-cable, and CAVEMEN-XML nodes representing them. Node names are listed within square-edge rectangles. *Name* attribute values are listed within rounded-edge rectangles. Letters aid in identifying what components correspond to *modelobject* nodes and what parts correspond to *subobject* and *atomicobject* nodes

For each lithium-ion battery, a part is the input socket for a lithium-ion battery cable. CAVEMEN-XML nodes representing the old lithium-ion battery are shown in Fig. 9.16d, e. The battery is specified using a *modelobject* node with the *name*attribute valued *mOldLiIonBattery* (Fig. 9.16d). The input socket for a lithium-ion battery cable is specified using an *atomicobject* node with the *name*attribute valued *aoOBBattCableInput*, where “OB” standards for “old lithium-ion battery” (Fig. 9.16e). The new lithium-ion battery is encoded in a similar way, replacing the *modelobject name*attribute value with *mNewLiIonBattery* and each instance of “OB” with “NB,” where “NB” stands for “new lithium-ion battery.”

For the Y-cable, parts include the two input sockets for a lithium-ion or lead battery cable and the output end connecting to a controller input socket. CAVEMEN-XML nodes representing the Y-cable are shown in Fig. 9.16f–i. The cable is specified using a *modelobject* node with the *name*attribute valued *mYCable* (Fig. 9.16f). The two input sockets are specified using *atomicobject* nodes with *name*attributes valued *aoYCableBatteryInput1* and *aoYCableBatteryInput2* respectively (Fig. 9.14g, h). The output end connecting to a controller input socket is specified using an *atomicobject* node with the *name*attribute *aoYCControllerOutput*, where "YC" stands for "Y-cable" (Fig. 9.14i).

9.7.1.2 Affordance Nodes

As mentioned, 13 affordances were encoded to represent the spatial relations among HES entities and end-user capabilities that are necessary to:

- Complete actions prescribed in the pump stopped alarm troubleshooting procedure (11 of 13 affordances)
- Connect the old lithium-ion battery to the new controller during the pump stopped alarm troubleshooting procedure (two of 13 affordances)

For the purpose of this case study, it is assumed that:

- All components are in-reach for the end user in all possible initial HES configurations (possible initial configurations discussed later)
- No HES entities are blocking connections (i.e., spatial relations involving connections being blocked are not specified, unlike the model in Chapter 6)

Three of 13 affordances involve the connector permanently attached to the heart; six of 13 involve connectability of a cable output end to an input socket; and four of 13 involve disconnectability of a cable output end from an input socket. All 13 affordances have one *humanoperator* child node with the *name*attribute valued *pPumpOperator* (i.e., the end user is the pump operator). The *affordance* nodes are described in the following sections. The end user's visual perspective and a subset of component configurations satisfying *relation* nodes are depicted in Figs. 9.17–9.22.

9.7.1.3 Affordances Involving the Permanently Attached Connector

The CAVEMEN-XML affordances involving the connector permanently attached to the heart include:

1. *ConnectorPartsAssemblable*, representing the end user's opportunity to reassemble the two connector parts that have broken and come apart

2. *ConnectorPartsDisassemblable*, representing the end user's opportunity to disassemble the two connector parts

3. *ConnectorPartsRotatable* representing the end user's opportunity to rotate the two connector parts 90°

All three affordances may need to be actualized multiple times to complete step-1a of the pump stopped alarm troubleshooting procedure. To actualize *ConnectorPartsAssemblable*, the two connector parts must be disassembled and the end user must be able to push them together by pushing one part back and the other part forth at the same time. The parts are disassembled if the part closest to the pump cable output end is not covering the front surface of the part farthest from the output end. The CAVEMEN-XML specification (Appendix H.1.2, lines 48–63) is described in outline form below:

- The part closest to the pump cable output end (*aoConnectorPart1*):
  - Cannot be covering the front surface of the other part (shown as disjoint in Fig. 9.17b)
  - And must be positionable back (end user's visual perspective of the pump cable, connector, and connector parts shown Fig. 9.17a)
- The other part (*aoConnectorPart2*), which is smaller and farthest from the output end, must be positionable forth

To actualize *ConnectorPartsDisassemblable*, the parts must be assembled and the end user must be able to pull them apart. The CAVEMEN-XML specification (Appendix H.1.2, lines 64–79) is described in outline form below:

- The part closest to the pump cable output end (*aoConnectorPart1*) must be:
  - Covering the front surface of the other part (as in Fig. 9.17c)
  - Positionable back (end user's visual perspective of the pump cable, connector, and connector parts shown Fig. 9.17c)
- The other part (*aoConnectorPart2*), which is smaller and farthest from the output end, must be positionable back

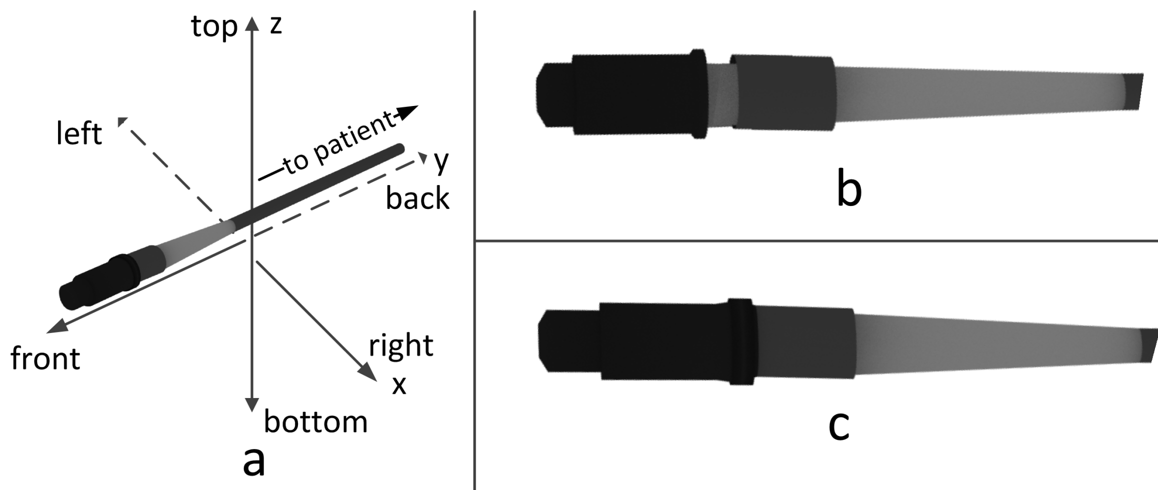


Figure 9.17: (a) Labeled axes show the pump operator's visual perspective for surfaces of the pump cable (*mPumpCable*). Surfaces are the same for the permanently attached connector (*sConnector*) and connector parts (*aoConnectorPart1*/*aoConnectorPart2*). (b–c) Graphical renderings of configurations satisfying *relation* nodes (*relation* nodes not shown). (b) *aoConnectorPart1* disjoint to *aoConnectorPart2*. (c) *aoConnectorPart1* covering front-side surface of *aoConnectorPart2*

To actualize *ConnectorPartsRotatable*, the parts must be disassembled and the end user must be able to rotate both parts. Because this affordance corresponds to the EOFM human action variable *hRotateParts*, which represents the end user rotating both parts at the same time, it is assumed that both parts must be rotatable rightward and leftward. The CAVEMEN-XML specification (Appendix H.1.2, lines 80–97) is described in outline form below:

- The part closest to the pump cable output end (*aoConnectorPart1*) must be:
  - Disjoint to the other part (as in Fig. 9.17b)

  - Rotatable right (end user's visual perspective shown in Fig. 9.17c)
  - Rotatable left

- The other part (*aoConnectorPart2*), which is smaller and farthest from the output end, must be:
  - Disjoint to the other part (as in Fig. 9.17b)
  - Rotatable right (end user's visual perspective shown in Fig. 9.17c)
  - Rotatable left

9.7.1.4 Affordances Involving Cable Disconnectability

As mentioned, four of 13 affordances involve cable disconnectability. Two of 13 CAVEMEN-XML *affordance* nodes specify disconnectability of the pump cable from the old controller. The other two of 13 specify disconnectability of a battery cable from the new controller.

The pump cable disconnectability affordances include:

1. *PumpCableDisconnectableFromAbCable* representing the end user's opportunity to disconnect the pump cable output end from the abdominal cable input socket
2. *PumpCableDisconnectableFromOldController* representing the end user's opportunity to disconnect the pump cable output end from the old controller input socket

Either one of these affordances needs to be actualized once to complete step-2a or step-2d of the pump stopped alarm troubleshooting procedure, depending on the initial pump cable configuration. To actualize *PumpCableDisconnectableFromAbCable*, the pump cable must be connected to the abdominal cable and the end user must be able to move both cable ends at the same time. The CAVEMEN-XML specification (Appendix H.1.2, 98–112) is described in outline form below:

- The pump cable output end (*aoPCControllerOutput*) must be positionable back (end user's visual perspective shown in Fig. 9.18b)
- The abdominal cable input socket (*apACPumpInput*) must be:

– Positionable back (end user's visual perspective shown in Fig. 9.18b)

– Covering all surfaces of the pump cable output end (as in Fig. 9.18d)

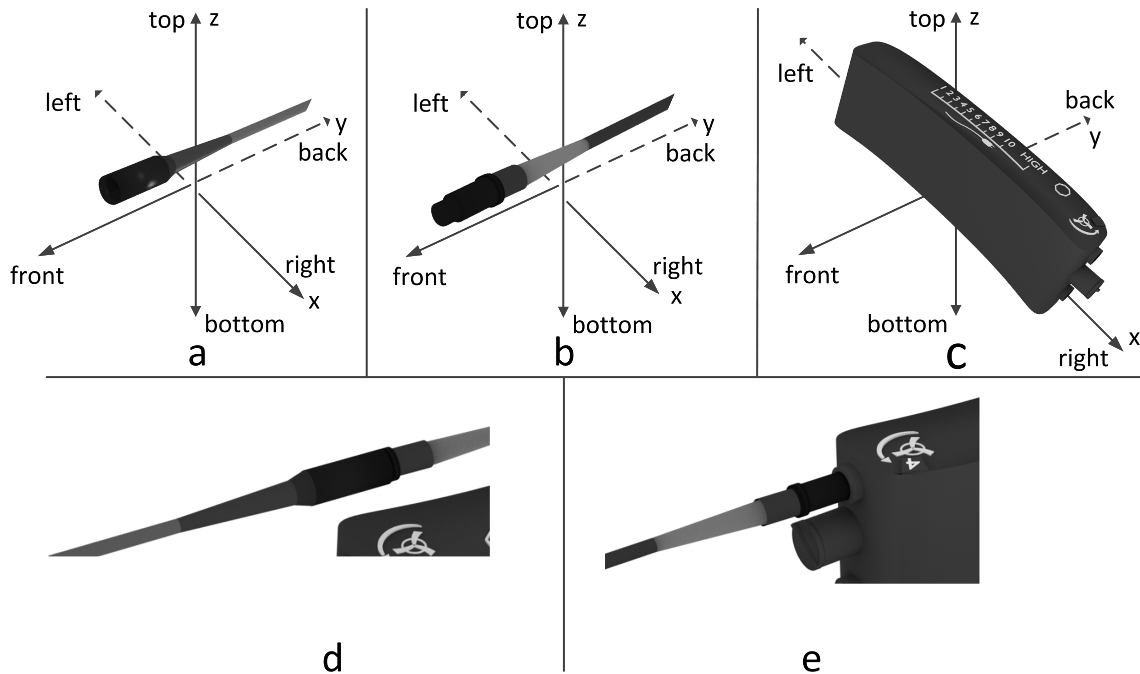


Figure 9.18: (a–c) Labeled axes show the pump operator's visual perspective for the: (a) abdominal cable (*mAbdominalCable*), (b) pump cable (*mPumpCable*), and (c) old controller (*mOldController*). (d–e) Graphical renderings of configurations satisfying *relation* nodes (*relation* nodes not shown). (d) Abdominal cable input socket (*aoACPumpInput*) covering all surfaces of the pump cable output end (*aoPCControllerOutput*). (e) Old controller pump cable input socket (*aoOCPumpInput*) covering all surfaces of the pump cable output end (*aoPCControllerOutput*)

To actualize *PumpCableDisconnectableFromOldController*, the pump cable must be connected to the old controller and the end user must be able to move the pump cable output end and old controller at the same time. The CAVEMEN-XML specification (Appendix H.1.2, 113–129) is described in outline form below:

- The old controller (*mOldController*) must be positionable back (end user's visual perspective shown in Fig. 9.18c)

- The pump cable output end (*aoPCControllerOutput*) must be positionable back (end user's visual perspective shown in Fig. 9.18b)

- The old controller pump cable input socket (*aoOCPumpCableInput*) must be covering all surfaces of the pump cable output end (as in Fig. 9.18e)

Affordances involving battery cable disconnectability include:

1. *LeadBattDisconnectableFromNewController* representing the end user's opportunity to disconnect the lead battery cable from the new controller
2. *NewLiBattCableDisconnectableFromNewController* representing the end user's opportunity to disconnect the new lithium-ion battery cable from the new controller

LeadBattDisconnectableFromNewController needs to be actualized to complete steps-4b of the pump stopped alarm troubleshooting procedure. To actualize *LeadBattDisconnectableFromNewController*, the lead battery cable must be connected to the new controller and the end user must be able to move the lead battery cable output end and new controller at the same time. The CAVEMEN-XML specification (Appendix H.1.2, 130–146) is described in outline form below:

- The new controller (*mNewController*) must be positionable back (end user's visual perspective shown in Fig. 9.19c)
- The lead battery cable output end (*aoLeadBattControllerOutput*) must be positionable back (end user's visual perspective of the lead battery cable output end shown in Fig. 9.19a)
- The new controller's battery cable input socket (*aoNCBatteryInput*) must be covering all surfaces of the lead battery cable output end (as in Fig. 9.19d)

NewLiBattCableDisconnectableFromNewController needs to be actualized to complete step-5(c)i of the pump stopped alarm troubleshooting procedure. To actualize *NewLiBattCableDisconnectable-FromNewController*, the new lithium-ion battery cable must be connected to the new controller and the end user must be able to move the cable output end and new controller at the same time. The CAVEMEN-XML specification (Appendix H.1.2, 147–163) is described in outline form below:

- The new controller (*mNewController*) must be positionable back (end user's visual perspective of the new controller shown in Fig. 9.19c)

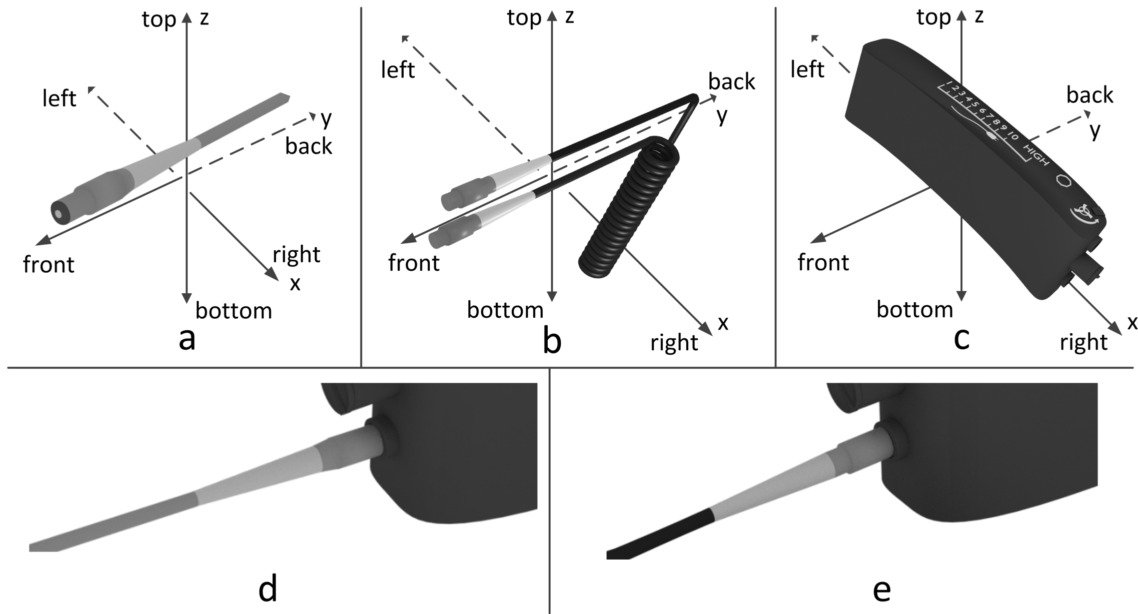


Figure 9.19: (a–c) Labeled axes show the pump operator's visual perspective for the: (a) lead battery cable output end (*aoLeadBattControllerOutput*) (b) new lithium-ion battery cable (*mNewLiBattCable*) and its output ends (*aoNBCControllerOutput*/*aoNBCControllerOutput*), and (c) new controller (*mNewController*). (d) New controller battery cable input socket (*aoNCBatteryInput*) covering all surfaces of the lead battery cable output end (*aoLeadBattControllerOutput*). (e) New controller battery cable input socket (*aoNCBatteryInput*) covering all surfaces of the new lithium-ion battery cable output end (*aoNBCControllerOutput*)

- The new lithium-ion battery cable output end (*aoNBCControllerOutput*) must be positionable back (end user's visual perspective shown in Fig. 9.19a)
- The new controller's battery cable input socket (*aoNCBatteryInput*) must be covering all surfaces of the new lithium-ion battery cable output end (as in Fig. 9.19e)

9.7.1.5 Affordances Involving Cable Connectability

As mentioned, six of 13 affordances involve cable connectability. One of 13 CAVEMEN-XML *affordance* nodes specifies connectability of the pump cable to a new controller; three of 13 specify connectability of a battery cable to the new controller; and two of 13 specify connectability of a battery cable to a lithium-ion battery.

The pump cable connectability affordance is *PumpCableConnectableToNewController*, representing the end user's opportunity to connect the pump cable to the new controller. The affordance needs to be actualized to complete step 3a of the pump stopped alarm troubleshooting procedure.

To actualize *PumpCableConnectableToNewController*, the pump cable cannot be connected to any input socket and the end user must be able to move the new controller and the pump cable output end at the same time. The CAVEMEN-XML specification (Appendix H.1.2, 164–198) is described in outline form below:

- The new controller (*mNewController*) must be positionable, translatable, and orientable along all axes (end user's visual perspective shown in Fig. 9.20b). This is because the new controller's pump cable input socket needs to be aligned with the pump cable output end
- For the same reason, the pump cable output end (*aoPCControllerOutput*) must be positionable, translatable, and orientable along all axes (end user's visual perspective shown in Fig. 9.20a)
- The abominable cable input socket, new controller pump cable input socket, and old controller pump cable input socket cannot be covering the pump cable output end (as in Fig. 9.20c)

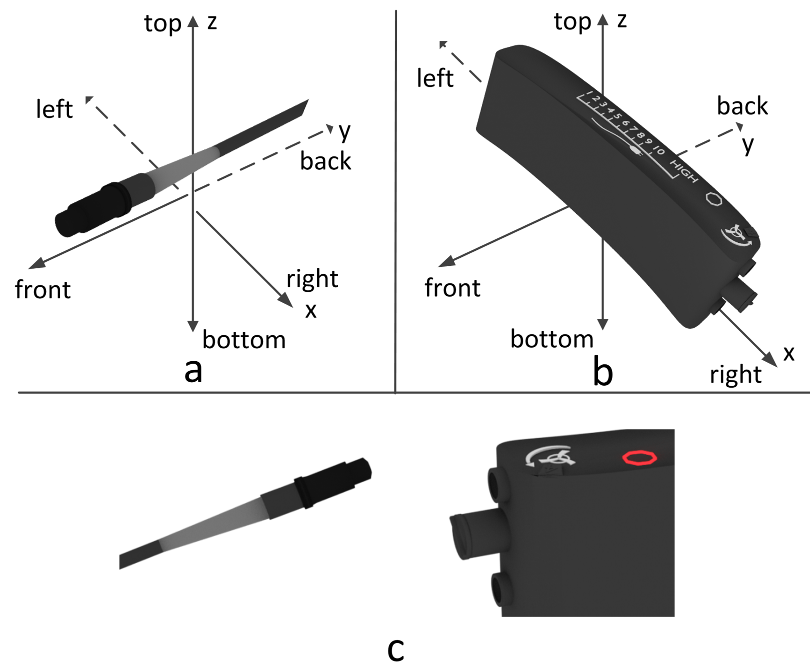


Figure 9.20: (a–b) Labeled axes show the pump operator's visual perspective for (a) the pump cable output end (*aoPCControllerOutput*) and (b) new controller (*mNewController*). (c) Pump cable output end is disjoint to all components

Battery cable-to-controller connectability affordances include:

1. *LeadBattConnectableToNewController*, representing the end user's opportunity to connect the lead battery cable to the new controller

2. *NewLiBattCableConnectableToNewController*, representing the end user's opportunity to connect the new lithium-ion battery cable to the new controller

3. *OldLiBattCableConnectableToNewController*, representing the end user's opportunity to connect the old lithium-ion battery cable to the new controller

LeadBattConnectableToNewController needs to be actualized to complete step-4a of the pump stopped alarm troubleshooting procedure. To actualize *LeadBattConnectableToNewController*, the lead battery cable cannot be connected to any input socket; the new controller's battery cable input socket cannot have a battery cable output end connected to it; and the end user must be able to move the lead battery cable output end and new controller at the same time. The CAVEMEN-XML specification (Appendix H.1.2, 199–243) is described in outline form below:

- The new controller (*mNewController*) must be positionable, translatable, and orientable along all axes (end user's visual perspective shown in Fig. 9.21b). This is because the new controller's battery cable input socket needs to be aligned with the lead battery cable output end

- For the same reason, the lead battery cable output end (*aoLeadBattControllerOutput*) must be positionable, translatable, and orientable along all axes (end user's visual perspective shown in Fig. 9.21a)

- The 1st Y-cable input socket (*aoYCableBatteryInput1*), 2nd Y-cable input socket (*aoYCableBatteryInput2*), new controller battery cable input socket (*aoNCBatteryInput*), and old controller battery cable input socket (*aoOCBatteryInput*) cannot be covering the lead battery cable output end (as in Fig. 9.21e)

- The new controller's battery cable input socket (*aoNCBatteryInput*) cannot be covering the old lithium-ion battery cable's controller output end (*aoOBCControllerOutput*), the new lithium-ion battery cable's controller output end (*aoNBCControllerOutput*), the lead battery cable output end (*aoLeadBattControllerOutput*), or the Y-cable output end (*aoYCControllerOutput*)

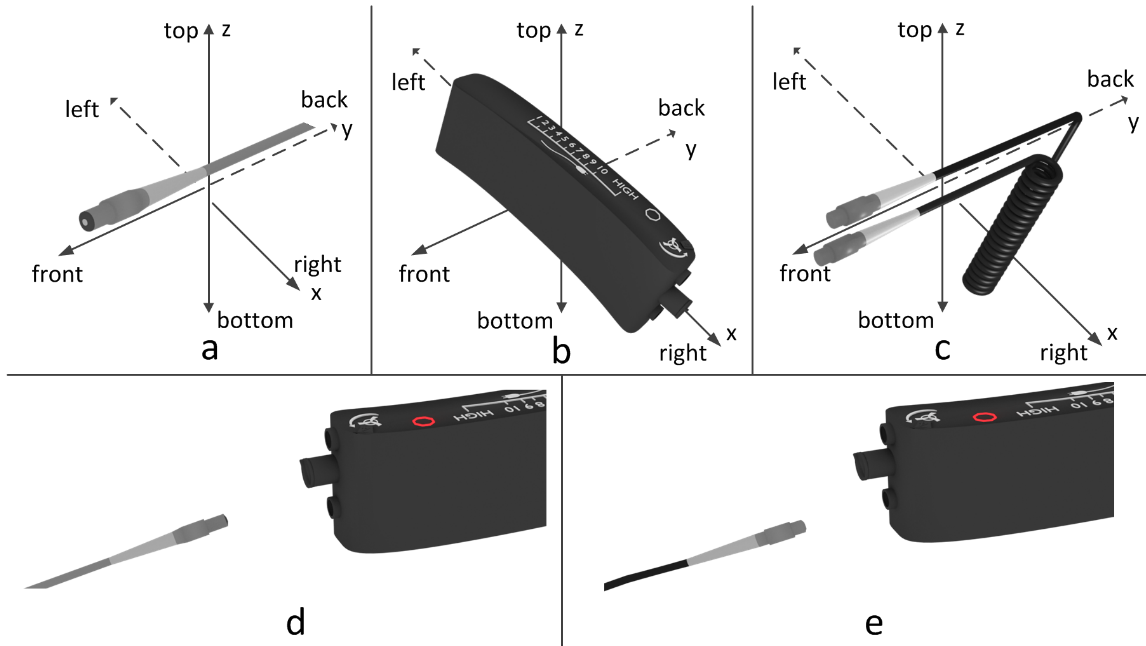


Figure 9.21: (a–c) Labeled axes show the pump operator's visual perspective for: (a) the lead battery cable output end (*aoLeadBattControllerOutput*), (b) the new controller (*mNewController*), and (c) the new lithium-ion battery cable output ends (*aoNBCControllerOutput/aoNBCBatteryOutput*) and old lithium-ion battery cable output ends (*aoOBCControllerOutput/aoOBCBatteryOutput*). (d) Lead battery cable output end disjoint to the controller's battery input socket. (e) New (or old) lithium-ion battery cable's controller output end disjoint to the new controller's battery input socket

NewLiBattCableConnectableToNewController needs to be actualized to complete step-5a of the pump stopped alarm troubleshooting procedure. To actualize *NewLiBattCableConnectableToNewController*, the new lithium-ion battery cable's controller output end cannot be connected to any input socket; the new controller's battery cable input socket cannot have a battery cable output end connected to it; and the end user must be able to move the new lithium-ion battery cable output end and the new controller at the same time. The CAVEMEN-XML specification (Appendix H.1.2, 244–288) is described in outline form below:

- The new controller (*mNewController*) must be positionable, translatable, and orientable along all axes (end user's visual perspective shown in Fig. 9.21b). This is because the new controller's battery cable input socket needs to be aligned with the new lithium-ion battery cable's controller output end
- For the same reason, the new lithium-ion battery cable's controller output end (*aoNBCCon-*

trollerOutput) must be positionable, translatable, and orientable along all axes (end user's visual perspective shown in Fig. 9.21a)

- The 1st Y-cable input socket (*aoYCableBatteryInput1*), 2nd Y-cable input socket (*aoYCableBatteryInput2*), new controller battery cable input socket (*aoNCBatteryInput*), and old controller battery cable input socket (*aoOCBatteryInput*) cannot be covering the new lithium-ion battery cable's controller output end (as in Fig. 9.21e)

- The new controller's battery cable input socket (*aoNCBatteryInput*) cannot be covering the old lithium-ion battery cable's controller output end (*aoOBCControllerOutput*), the new lithium-ion battery cable's controller output end (*aoNBCControllerOutput*), the lead battery cable output end (*aoLeadBattControllerOutput*), or the Y-cable output end (*aoYCControllerOutput*)

OldLiBattCableConnectableToNewController should not be actualized if the old lithium-ion battery cable was in-use when the pump stopped alarm engaged. To actualize *OldLiBattCableConnectableToNewController*, the old lithium-ion battery cable's controller output end cannot be connected to any input socket; the new controller's battery cable input socket cannot have a battery cable output end connected to it; and the end user must be able to move the old lithium-ion battery cable output end and new controller at the same time. The CAVEMEN-XML specification (Appendix H.1.2, 289–333) is similar to *NewLiBattCableConnectableToNewController*, but with two modifications:

- The old lithium-ion battery cable's controller output end (*aoOBCControllerOutput*, replacing *aoNBCControllerOutput*) must be positionable, translatable, and orientable along all axes (end user's visual perspective shown in Fig. 9.21a)

- The 1st Y-cable input socket (*aoYCableBatteryInput1*), 2nd Y-cable input socket (*aoYCableBatteryInput2*), new controller battery cable input socket (*aoNCBatteryInput*), and old controller battery cable input socket (*aoOCBatteryInput*) cannot be covering the old (instead of new) lithium-ion battery cable's controller output end (as in Fig. 9.21e)

Lithium-ion battery cable-to-battery connectability affordances include:

1. *NewLiBattCableConnectableToNewLiBatt*, representing the end user's opportunity to connect the new lithium-ion battery cable to the new lithium-ion battery

2. *NewLiBattCableConnectableToOldLiBatt*, representing the end user's opportunity to connect the new lithium-ion battery cable to the old lithium-ion battery

NewLiBattCableConnectableToNewLiBatt needs to be actualized to complete step-5b of the pump stopped alarm troubleshooting procedure. To actualize *NewLiBattCableConnectableToNewLiBatt*, the new lithium-ion battery cable's battery output end cannot be connected to a lithium-ion battery and the end user must be able to move the battery cable output end and the battery at the same time. The CAVEMEN-XML specification (Appendix H.1.2, 334–365) is described in outline form below:

- The new lithium-ion battery (*mNewLiIonBattery*) must be positionable, translatable, and orientable along all axes (end user's visual perspective shown in Fig. 9.22b). This is because the new lithium-ion battery's input socket needs to be aligned with the new lithium-ion battery cable's battery output end

- For the same reason, the new lithium-ion battery cable's battery output end (*aoNBCBattery-Output*) must be positionable, translatable, and orientable along all axes (end user's visual perspective shown in Fig. 9.22a)

- The old lithium-ion battery input socket (*aoOBBatteryInput*) cannot be covering the new lithium-ion battery cable's battery output end (as in Fig. 9.22c)

- The new lithium-ion battery input socket (*aoNBBatteryInput*) cannot be covering the old lithium-ion battery cable's battery output end (*aoOBCBatteryOutput*) or the new lithium-ion battery cable's battery output end (as in Fig. 9.22c)

NewLiBattCableConnectableToOldLiBatt should not be actualized if the old lithium-ion battery was in-use when the pump stopped alarm engaged. To actualize *NewLiBattCableConnectable-ToOldLiBatt*, the new lithium-ion battery cable's battery output end cannot be connected to a

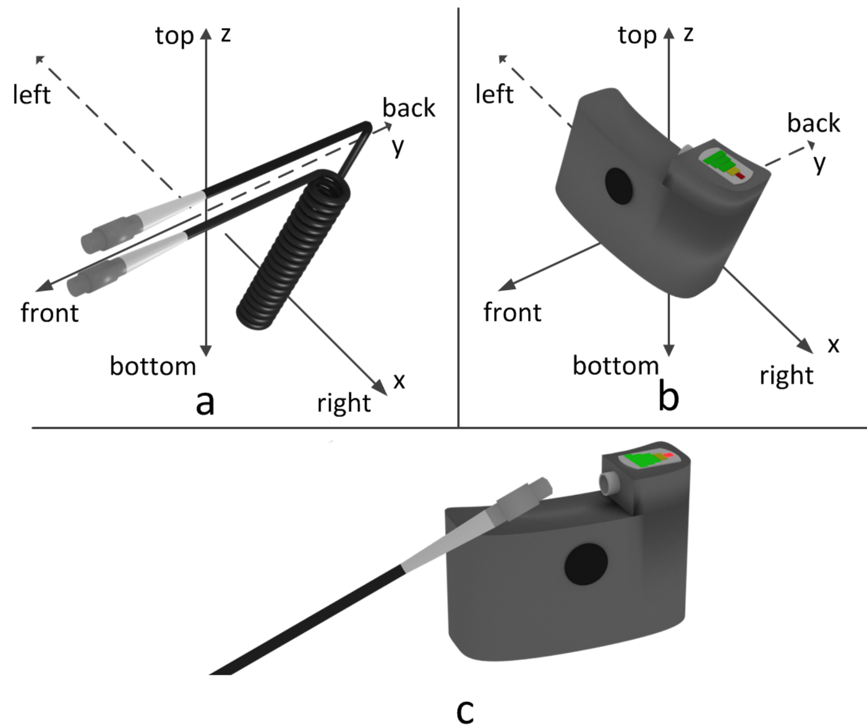


Figure 9.22: (a, b) Labeled axes show the pump operator's visual perspective for the: (a) new lithium-ion battery cable output ends (*aoNBCControllerOutput*/*aoNBCBatteryOutput*) and old lithium-ion battery cable output ends (*aoOBCControllerOutput*/*aoOBCBatteryOutput*), (b) new lithium-ion battery (*mNewLiIonBattery*) and old lithium-ion battery (*mOldLiIonBattery*). (c) New (or old) lithium-ion battery cable's battery output end disjoint to new controller's battery input socket

lithium-ion battery and the end user must be able to move the battery cable output end and the battery at the same time. The CAVEMEN-XML specification (Appendix H.1.2, 366–397) is similar to *NewLiBattCableConnectableToNewLiBatt*, but with two modifications:

- The old lithium-ion battery (*mOldLiIonBattery*, instead of *mNewLiIonBattery*), must be positionable, translatable, and orientable along all axes (end user's visual perspective shown in Fig. 9.22b)

- The old lithium-ion battery input socket (*aoOBBatteryInput*, instead of *aoNBBatteryInput*), cannot be covering the old lithium-ion battery cable's battery output end (*aoOBCBatteryOutput*) or the new lithium-ion battery cable's battery output end (as in Fig. 9.21e)

9.7.2 HES Module

As mentioned, the automatically generated HES module was augmented with manually encoded infrastructure to specify:

- Static end-user motor capabilities
- Evolving spatial relations among human-environment system (HES) entities

The 26 *ability* nodes of the CAVEMEN-XML representation were translated to 11 output variables having the record type `abilities` (i.e., the translator automatically removed duplicate nodes). For all 11 variables, initializations were encoded to specify that the end user can move all components along all six degrees of freedom (Appendix H.2.3, lines 284–327).

The 38 *relation* nodes of the CAVEMEN-XML representation were translated to 11 output variables having corresponding record types (Appendix H.2.3, lines 260–270). Nine of 11 represent input sockets and two of 11 represent parts of the permanently attached connector. Initial and next-state spatial relations were assigned using the SAL `DEFINITION` construct, conditional expressions, and EOFM input variables leveraged from the pump stopped alarm troubleshooting procedure *task* model. For each of the 13 EOFM input variables representing a cable connection (e.g. *iPumpCableToOldController*), a value of "*Connected*" corresponds to an input socket covering all surfaces of a cable output end. Otherwise, a value of "*Disconnected*" corresponds to an input socket disjoint to all surfaces of a cable output end. The SAL syntax encoded generally below was utilized to assign nine of 11 spatial relations using the 13 EOFM input variables representing cable connections.

```
aoInputSocket =
IF iComponentToSource_1 = Connected THEN
   (#aoOutputEnd_1 := [[x:  directional]covering],
    aoOutputEnd_2 := [[x:  directional]disjoint_to],
    ...
    aoOutputEnd_N := [[x:  directional]disjoint_to]#)
ELSIF ...
ELSIF iComponentToSource_N = Connected THEN
  (#aoOutputEnd_1 := [[x:  directional]disjoint_to],
    aoOutputEnd_2 := [[x:  directional]disjoint_to],
    ...
    aoOutputEnd_N := [[x:  directional]covering]#)
ELSE
  (#aoOutputEnd_1 := [[x:  directional]disjoint_to],
    aoOutputEnd_2 := [[x:  directional]disjoint_to],
    ...
    aoOutputEnd_N := [[x:  directional]disjoint_to]#)
ENDIF;
```

Two of 11 spatial relations were assigned using the EOFM input variable representing the connector permanently attached to the heart (*iPermanentlyAttachedConnector*). A value of "*Assembled*" corresponds to *aoConnectorPart1* covering the front-side surface of *aoConnectorPart2* and *aoConnectorPart2* covering the back-side surface of *aoConnectorPart1*. Otherwise, a value of "*Broken*" corresponds to all surfaces of both parts disjoint to each other. The SAL syntax representing these relations is provided in Appendix H.2.3, lines 465–483.

9.8 Device

The *device* model is composed of:

1. One *end user-device interaction* model (described in Section 9.8.1)
2. One *display/control logic* model (described in Section 9.8.2)
3. One *plant* model (described in Section 9.8.3)
4. One *constraints* model (described in Section 9.8.3)

9.8.1 End User-Device Interaction

The end user-device interaction model is a modified version of the model from Chapter 5. Because configurable hardware was not modified in the redesigned interface, initial configuration assignments

were specified using the same SAL code (Appendix H.2.5, lines 66–111). 18 guarded transitions were encoded. 16 of 18 correspond to human actions represented in the pump stopped alarm troubleshooting procedure *task* model. Two of 18 correspond to actions represented in the pump speed adjustment procedure *task* model (Fig. 9.23d):

1. *hRotateKnobClockwise*

2. *hRotateKnobCounterClockwise*

11 of 18 transitions have corresponding affordances represented in the *affordance* model, where 11 *affordance* model variables correspond to respective EOFM human actions. In this case study, one action variable and one affordance variable are leveraged within conditional expressions of the *end user-device interaction* model controlling whether each action effects changes to configurable hardware (Fig. 9.23a–c). Eight cable connection/disconnection transitions were encoded individually; however, because they were encoded in a similar way, all eight are depicted in Fig. 9.23a. Cable connection/disconnection affordances are encoded generally as *CableToSourceConnectable* and *CableToSourceDisconnectable* respectively. Two guarded transitions represent the end user disassembling and reassembling the connector permanently attached to the heart (Fig. 9.23c). One guarded transition represents the end user rotating parts of the connector permanently attached to the heart (Fig. 9.23d).

SAL code of the end user-device interaction model is provided in Appendix H.2.5, lines 9–155.

9.8.2 Display/Control Logic

The *display/control logic* model represents the controller's algorithms for illuminating lights and emitting audible alerts. Leveraging the syntax of EOFM, one output variable represents what alarm is engaged on the controller (*iAlarm*) and one output variable represents what power indicator light is illuminated (*iPowerLight*). A third, Boolean output variable represents whether the system is in a functional configuration (*functional*). This variable was encoded to demonstrate an implementation of the framework integrating configurable hardware and the target system's control logic.

Guarded initializations for *iAlarm* and *iPowerLight* are represented graphically in Fig. 9.24,

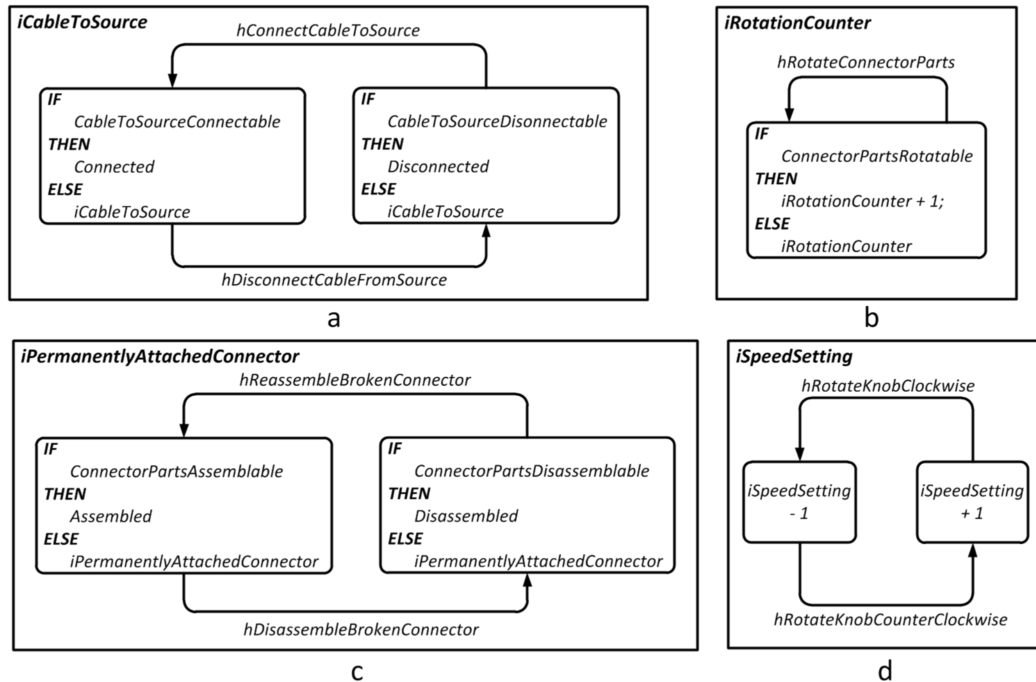


Figure 9.23: Visual representation of transitions encoded in the *end user-device interaction* model. (a) General encoding of the eight cable connection/disconnection transitions involving end-user connection/disconnection actions and connectable/disconnectable affordances. (b) Transitions involving the end-user action of rotating connector parts and the affordance of part rotatability. (c) Transition involving the end-user actions of assembling/disassembling parts of the permanently attached connector and assemblable/disassemblable affordances (d) Transitions to the speed setting knob. The EOFM human action variable *hRotateKnobCounterClockwise* represents the end user rotating the knob in the direction of the curved arrow to increase the speed setting by one. The variable *hRotateKnobClockwise* represents the end user rotating the knob in the opposite direction to decrease the speed setting by 1

where *speed* and *power* are outputs of the *plant* model (discussed later). The initial state of *iAlarm* is:

- *HighPower* if the pump speed is greater than 0 and the power supplied to the pump is greater than 3.5 watts
- *PumpStopped* if the pump speed is 0
- *NoAlarm* otherwise

The initial state of *iPowerLight* is:

- 0 if the power supplied to the pump less than or equal to 0.05 watts

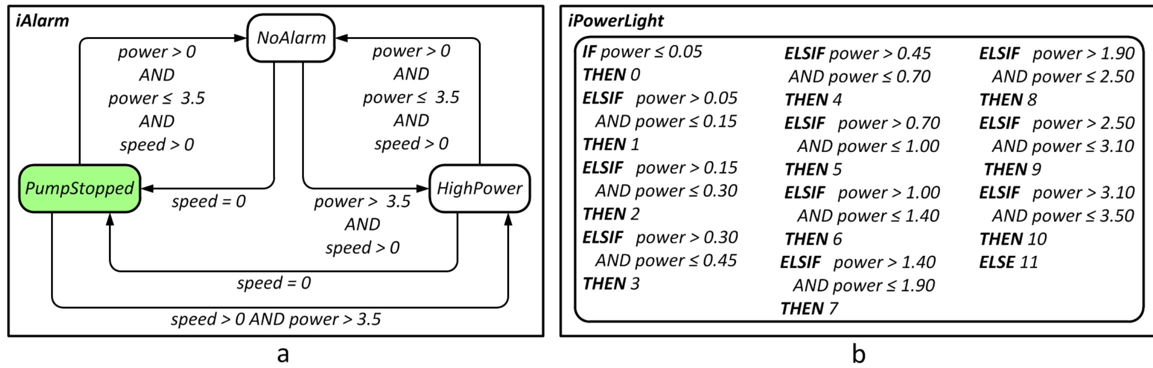


Figure 9.24: Graphical representation of guarded initializations of the *display/control logic* model. Variable names are listed in boldface, italic text within square-edge rectangles. Variable values are listed in italic text within rounded-edge rectangles. Arrow labels in italic text are valued input variables from the *plant* model. (a) Transitions for the variable *iAlarm* representing what alarm is engaged on the controller. (b) Transitions for the variable *iPowerLight* representing what indicator light is illuminated on the controller. The value "*0*" corresponds to no lights illuminated. The values *1–10* correspond to lights having numeric labels 1–10. The value "*11*" corresponds to the light labeled "*HIGH*"

- 1–10 is the power supplied to the pump falls within corresponding ranges identified in Table 9.1
- 11 otherwise (power supplied is greater than or equal to 3.5 watts)

Guarded transitions for *iAlarm* and *iPowerLight* have the same semantics as guarded initializations, replacing each instance of *speed* and *power* in Fig. 9.24 with *speed'* and *power'*, where an apostrophe means "in the next-state." These semantics ensure that the indicator lights and audible alarms reflect the current pump speed and power supplied. SAL syntax is provided in Appendix H.2.5, lines 198–293.

The Boolean output variable *functional* represents whether hardware is configured and functioning in a way that enables normal pump operation. As mentioned in Section 9.3, a functional configuration is one in which a power supply is connected to the controller, the pump is connected to the controller, and the permanently attached connector is assembled. In such a configuration, the components may appear to be functioning normally; however, an unobservable malfunction could occur and interrupt the power supply, causing the pump stopped alarm to engage. When *functional* is valued *true*, this means that the hardware is in a functional configuration and there are no malfunctions. When *functional* is valued *false*, this means there is a malfunction or hardware is not in a

functional configuration. Thus, considering possible initial and next-state hardware configurations represented in the *end user-device interaction* model, *functional* can be either *true* or *false* if:

- The pump cable is connected in one of the following configurations:
  - Connected directly to the old controller
  - Connected to the old controller via the abdominal cable
  - Connected directly to the new controller
- A power supply is connected in one of the following configurations:
  - The lead battery is connected directly to the old controller
  - The lead battery is connected to the old controller via the Y-cable
  - The old lithium-ion battery is connected directly to the old controller
  - The old lithium-ion battery is connected to the old controller via the Y-cable
  - The lead battery is connected to the new controller
  - The new lithium-ion battery is connected directly to the new controller
- The permanently attached connector is assembled

Otherwise, *functional* is *false*. This was encoded in SAL using the `DEFINITION` construct, an `IN` selection statement, a conditional expression, and EOFM input variables (Appendix H.2.5). These semantics enable the pump stopped alarm to:

- Engage in all possible hardware configurations represented in the model
- Disengage during the pump stopped alarm troubleshooting procedure

9.8.3 Plant and Constraints

The *plant* and *constraints* models representing continuous pump dynamics are modified versions of the models from Chapter 8. Two modifications were implemented for this case study:

1. Guarded transitions in the *plant* model incorporate the variable `functional` as an input from the *display/control logic* model: speed, power, and flow can only be greater than 0 if the human-system interface is in a functional configuration and there are no malfunctions (i.e., *functional* is valued *true*)

2. Guarded transitions in the *constraints* model enable `fitsData` to be true if speed, power, and flow are equal to 0, which enables the pump stopped alarm to engage while the device is operating within data-constrained parameters (i.e., 0 watts supplied to the pump, a pump speed of 0 RPM, and a flow rate of 0 LPM will be considered within model checking analyses)

These models were 260 lines of SAL code (Appendix H.2.6).

9.9 Signifiers

Utilizing the BIGSIS approach, a formal signifier model was encoded to represent what is signified by the power indicator lights (including the high power alarm), the pump stopped alarm, and the speed setting knob. The model represents visual properties of color and label, audible properties of pattern and volume, and explanations of what is signified within accompanying documentation. The BIGSIS-XML model (described in Section 9.9.1) is a modified version of the model from Chapter 7. The BIGSIS-SAL model (described in Section 9.9.2) was automatically translated and augmented with additional model infrastructure controlling:

- Initial- and next-states of end-user descriptions via the exchange of input/output variables with the *display/control logic* model. These semantics ensure that signifiers are updated to reflect the current-state of the device

- What is signified through the documentation channel based on the variable *iPage* from the *documentation navigation* model. These semantics demonstrate an implementation of the framework in which a documented device description is only known when the end user is on the page containing it (i.e., pages of the patient handbook behave like screens of a graphical display)

9.9.1 BIGSIS-XML Model

18 words/phrases were encoded to represent what is signified by the power indicators, pump stopped alarm, and speed setting knob; their colors, labels, audible patterns, and volumes; and accompanying documentation describing what is signified. Six of 18 words were encoded as text content within a *signified-meanings* node named *PumpSpeed*, all of which define relative pump speeds signified by the visual/audible properties of the pump stopped alarm and visual properties of the speed setting knob: *Stopped*, *Low*, *Lowest*, *Medium*, *High*, and *Highest*.

12 of 18 phrases were encoded as text content within a *signified-meanings* node named *PowerSupplied*. 11 of 12 words define power supplied in units signified by visual/audible properties of the pump stopped alarm and power indicators, excluding the high power alarm: *ZeroUnits*, *OneUnit*, *TwoUnits*, *ThreeUnits*, *FourUnits FiveUnits*, *SixUnits*, *SevenUnits*, *EightUnits*, *NineUnits*, and *TenUnits*. One word defines relative power supplied signified by the high power alarm: *TooHigh*.

Signifier-properties, *property-documentation*, *Color*, *Label*, and *aPattern* nodes of the BIGSIS-XML representation are described in outline form. The first outline describes what is signified through visual and audible channel properties of interface components described in Section 9.3, and the second outline describes what is signified through the documentation channel. To aid in associating outlined descriptions of what is signified with model variable names in the BIGSIS-XML representation, *signifier-properties* and *property-documentation* node *of* attributes are listed in italic text within parentheses.

The first outline describing what is signified by colors, labels, audible patterns, and volumes presented on the device is shown below.

1. Power indicators (*signifier-properties of="PowerIndicators"*)

   (a) *Color*: amber signifies too many power units supplied to the pump (*TooHigh*). Green signifies power units supplied by the pump, depending on what label is colored

   (b) *Label*: a number labeled 1–10 signifies a corresponding power unit supplied to the pump (e.g. a label described as "*One*" signifies a power supplied of "*OneUnit*"). The label

"HIGH" signifies too many power units supplied to the pump (*TooHigh*)

(c) *Volume*: a loud volume signifies power supplied to the pump, depending on what audible pattern is emitted

(d) *aPattern*: the phrase "POWER TOO HIGH" emitted periodically signifies too many power units supplied to the pump (*TooHigh*)

2. Pump stopped alarm (*signifier-properties of="PumpStoppedAlarm"*)

   (a) *Color*: red signifies that the pump is stopped. When no color is present, signified pump speed depends on the speed setting knob's label

   (b) *Volume*: a loud volume signifies pump speed, depending on what audible pattern is emitted

   (c) *aPattern*: the phrase "PUMP STOPPED" emitted periodically signifies that the pump is stopped

3. Speed setting knob (*signifier-properties of="SpeedSettingKnob"*)

   (a) *Color*: White signifies pump speed, depending on what label is colored. No color signifies pump speed, depending on the color of the pump stopped alarm

   (b) *Label*: a number 1–5 signifies relative pump speed, lowest–highest. No label signifies pump speed, depending on the color of the pump stopped alarm

The second outline describing what is signified through the documentation channel is shown below.

1. Documentation of power indicators (*property-documentation of="PowerIndicators"*)

   (a) *Color*: amber signifies too many power units supplied to the pump (*TooHigh*). Green signifies power units supplied by the pump, depending on what label is colored. No color signifies that the pump is stopped and zero power units are supplied to the pump

   (b) *Label*: a number labeled 1–10 signifies a corresponding power unit supplied to the pump. The label "HIGH" signifies too many power units supplied to the pump (*TooHigh*). No label signifies zero power units supplied to the pump

   (c) *Volume*: a loud volume signifies power supplied to the pump, depending on what audible pattern is emitted

   (d) *aPattern*: the phrase “POWER TOO HIGH” emitted periodically signifies too many power units supplied to the pump (*TooHigh*)

2. Documentation of pump stopped alarm (*property-documentation of=“PumpStoppedAlarm”*)

   (a) *Color*: red signifies that the pump is stopped. When no color is present, signified pump speed depends on the speed setting knob’s label

   (b) *Volume*: a loud volume signifies pump speed, depending on what audible pattern is emitted

   (c) *aPattern*: the phrase “PUMP STOPPED” emitted periodically signifies that the pump is stopped

3. Documentation of speed setting knob (*property-documentation of=“SpeedSettingKnob”*)

   (a) *Color*: White signifies pump speed, depending on what label is colored. No color signifies pump speed, depending on the color of the pump stopped alarm

   (b) *Label*: a number 1–5 signifies relative pump speed, lowest–highest. No label signifies pump speed, depending on the color of the pump stopped alarm

The BIGSIS-XML representation was encoded in 76 lines (Appendix H.1.3).

9.9.2 BIGSIS-SAL Model

The BIGSIS-XML representation was translated to 86 lines of SAL code using the JavaScript-based translation tool described in Chapter 7. As mentioned, additional model infrastructure was manually encoded to represent:

- Initial- and next-state end-user descriptions via the exchange of input/output variables with the *display/control logic* model

- What is signified through the documentation channel based on the value of *iPage* from the *documentation navigation* model

9.9.2.1 End-User Descriptions

Guarded transitions to end-user descriptions are depicted graphically in Figs. 9.25–9.27. Guarded initializations were encoded in the same way by removing apostrophes from variables represented in Figs. 9.25–9.27. These semantics ensure that initial- and next-states of end-user description reflect the current-state of the device.

For color of the power indicators (Fig. 9.25a), the *display/control logic* model output variable *iPowerLight* controls the end-user description. This is because a power light could be illuminated green or distinguished when the pump stopped alarm is engaged, depending on whether hardware is in a functional configuration and there are no malfunctions. For colors of the speed setting knob and pump stopped alarm (Fig. 9.25b, c), the *display/control logic* model output variable *iAlarm* controls end-user descriptions. As mentioned, the speed setting knob label backlight is extinguished and the pump stopped alarm light is illuminated red when the pump stopped alarm is engaged. Otherwise, the speed setting knob label is back-lit white and the pump stopped alarm light is extinguished.

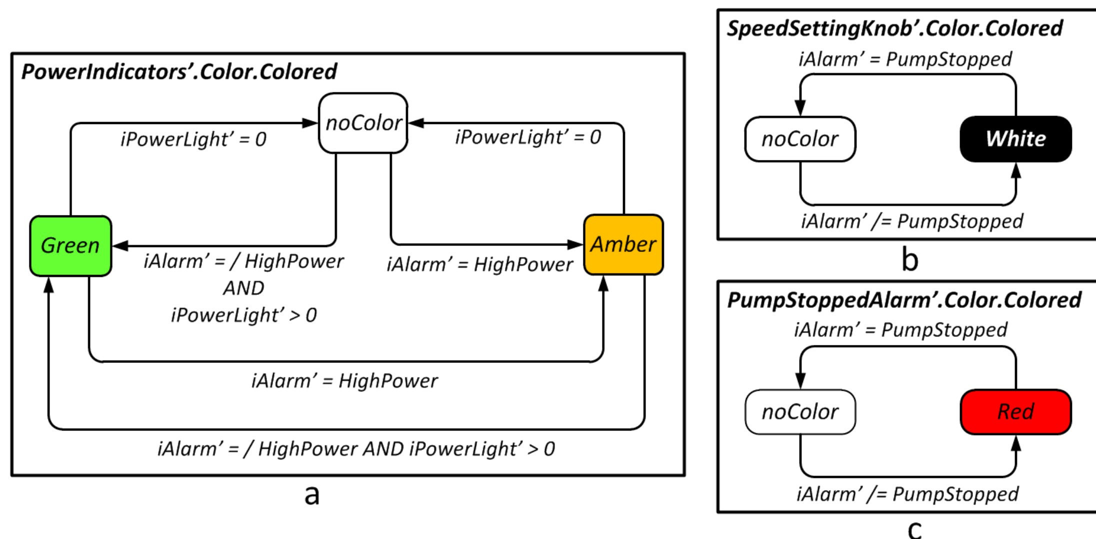


Figure 9.25: Graphical representation of guarded transitions to end-user descriptions of color. Variable names are listed in boldface, italic text within square-edge rectangles. Variable values are listed in italic text within rounded-edge rectangles. Arrow labels in italic text are valued variables from the *display/control logic* model. (a) End-user description of color for the power indicators. (b) End-user description of color for the speed setting knob. (c) End-user descriptions of color for the pump stopped alarm

Similar to the color, end-user description of label for the power indicators (Fig. 9.26a) is controlled

by the *display/control logic* model output variable *iPowerLight*. Utilizing guarded transitions and conditional expressions, end-user description of label for the speed setting knob (Fig. 9.26b) is controlled by:

1. The *display/control logic* model output variable *iAlarm*

2. The *end user-device interaction* model output variable *iSpeedSetting*

For the speed setting knob, an end-user description of "*noLabel*" is assigned when the pump stopped alarm is engaged; otherwise, if the pump stopped alarm is not engaged, the end-user description of label reflects the current speed setting.

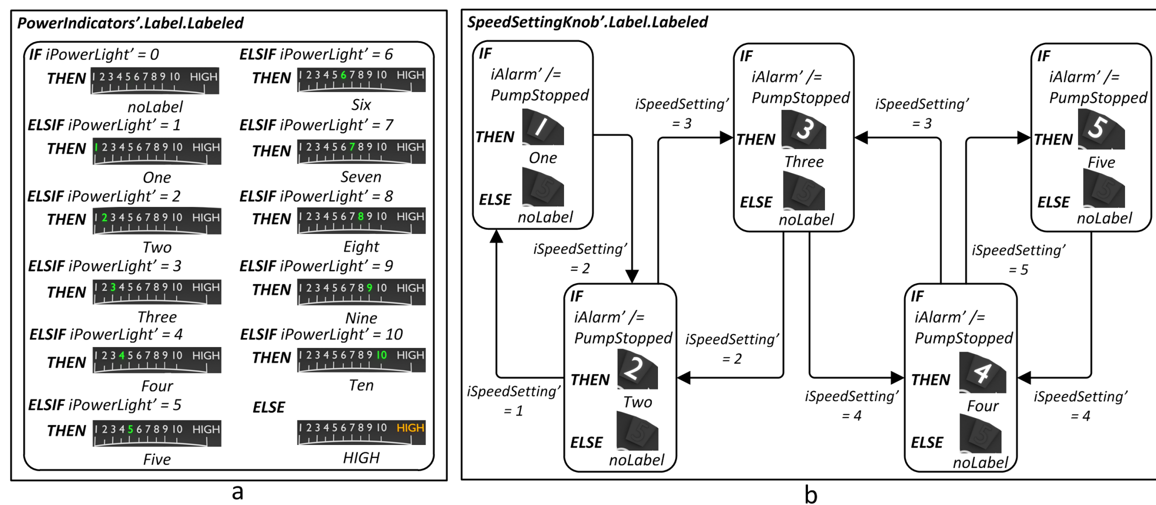


Figure 9.26: Graphical representation of guarded transitions to end-user descriptions of label. Variable names are listed in boldface, italic text within square-edge rectangles. Variable values are listed in italic text within rounded-edge rectangles. Arrow labels in italic text are valued variables from the *display/control logic* or *end user-device interaction* models. (a) End-user description of label for the power indicators. (b) End-user description of label for the speed setting knob

When an alarm is engaged, an audible alert is only emitted by a controller if the 1/2 cell alarm battery cap is tightened. Because one controller can be in-use at a time, end-user descriptions of visual properties are derived from one controller. However, if both controllers are emitting audible alarms at the same time, the end user may be unable to determine what controller is emitting the alarm. Thus, in this case study, end-user descriptions of audible properties (Fig. 9.27) are controlled by:

1. The *display/control logic* model output variable *iAlarm*

2. The *end user-device interaction* model output variable *iOldControllerABCap*

3. The *end user-device interaction* model output variable *iNewControllerABCap*

For both audible alarms, the end-user descriptions of volume can be "*Loud*" if either 1/2 cell AA alarm battery cap is tightened (Fig. 9.27a, b). Since both alarms are emitted at the same volume, the level is considered the same if one or both controllers are emitting different alarms (i.e., the possible, relatively minor increase in volume does not constitute a new identifiable volume). However, the end-user description of audible pattern can be "*PUMP_STOPPED*" or "*POWER_TOO_HIGH*" if exactly one 1/2 cell AA alarm battery cap is tightened (Fig. 9.27c, d). If both 1/2 cell AA battery caps are tightened, two different audible patterns could be emitted ("high power" for the controller in-use and "pump stopped" for the other controller). Thus, there could be multiple inputs to the *relation* and *explanation* functions, resulting in nothing being signified. If both controllers are emitting "pump stopped," they could be asynchronous, also resulting in multiple inputs to the *relation* and *explanation* functions. Thus, in this case study, exactly one alarm battery cap must be tightened for there to be a one-to-one relation between audible pattern and signified meaning. The possibility of both controllers emitting "pump stopped" synchronously is not considered.

The SAL code for end-user descriptions is provided in Appendix H.2.4. Lines 36–102 are initializations and lines 125–191 are transitions.

9.9.2.2 Documentation Channel

Automatically generated model infrastructure controlling what is signified through the documentation channel was modified to specify that what is signified depends on:

1. End-user descriptions of perceivable device properties

2. What page of the patient handbook the end user is on

Such an encoding demonstrates an implementation of the framework in which pages of the patient handbook behave similarly to screens of a graphical display. The modified SAL syntax for representing documentation-channel signifiers in this way is encoded generally below. For initializations,

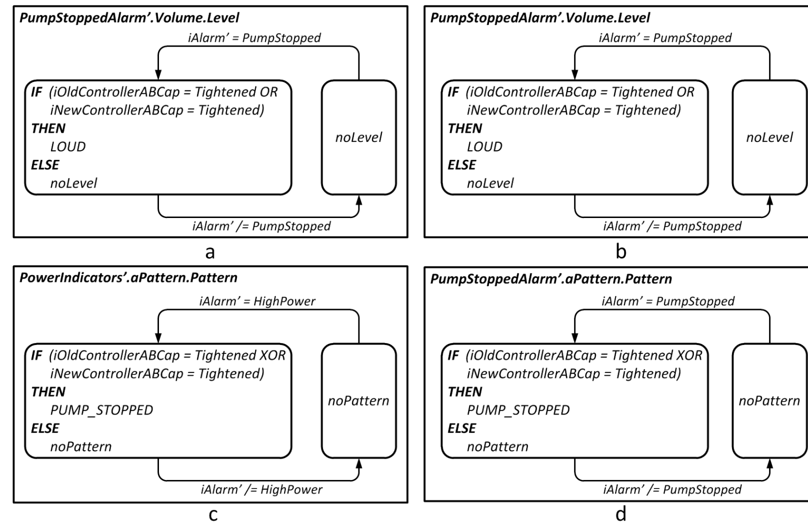


Figure 9.27: Graphical representation of guarded transitions to end-user descriptions of volume and audible pattern. Variable names are listed in boldface, italic text within square-edge rectangles. Variable values are listed in italic text within rounded-edge rectangles. Arrow labels in italic text are valued variables from the *display/control logic* or *end user-device interaction* models. (a) End-user description of volume for the pump stopped alarm. (b) End-user description of volume for the power indicators. (b) End-user description of audible pattern for the pump stopped alarm. (d) End-user description of volume for the power indicators

conditional expressions are augmented with the current-value of *iPage*. For transitions, conditional expressions are augmented with the next-state value of *iPage*. These semantics specify that a function or meaning is signified through the documentation channel if the end-user description of a device property relates to the explanation provided in documentation and the end-user is on the page containing that explanation.

```
INITIALIZATION
 Doc_Component.Property.Category =
  IF iPage = page_1 AND Component.Property.Description = description_1
   THEN signified_1
  ELSIF iPage = page_2 AND Component.Property.Description = description_2
   THEN ...ELSIF ...THEN ...
  ELSE CategoryNotSignified;

TRANSITION
 Doc_Component'.Property.Category =
  IF iPage = page_1 AND Component'.Property.Description = description_1
   THEN signified_1
  ELSIF iPage = page_2 AND Component'.Property.Description = description_2
   THEN ...ELSIF ...THEN ...
  ELSE CategoryNotSignified;
```

For the case study *signifier* model, page numbers controlling what is signified through the

documentation channel are described in outline form below:

1. Page-8 explains what is signified by:
   - Power indicators colored green and labeled 1–10
   - The speed setting knob colored white and labeled 1–5
2. Page-10 explains what is signified by:
   - Power indicators colored amber and labeled “HIGH”
   - Power indicators having no color or label
   - The pump stopped alarm colored red
   - The speed setting knob having no color or label
   - The pump stopped alarm emitting the audible pattern “pump stopped” at a loud volume
   - Power indicators emitting the audible pattern “power too high” at a loud volume

The SAL code for documentation-channel signifiers is provided in Appendix H.2.4. Lines 112–121 are initializations and lines 192–210 are transitions.

9.10 System Model Composition

A system model was composed to demonstrate one implementation of the integrated framework model architecture. In this case study, the *documentation navigation model* transitions sequentially and asynchronously with other models, as it represents an end user interacting with just the document and no other elements of the interface concurrently. Each *task model* also transitions sequentially and asynchronously with other models, as the end user cannot concurrently execute procedures listed on different pages in documentation. All other models transition concurrently and synchronously with each other: *signifier* (Fig. 9.1c), *affordance* (Fig. 9.1d), *plant* (Fig. 9.1f), *constraints* (Fig. 9.1g), *display/control logic* (Fig. 9.1h), and *end user-device interaction* (Fig. 9.1i). Such a transition protocol ensures that all models representing the device, what is signified to the end user, and affordances emergent in the operational environment respond correctly to human action outputs of the *task* model(s) and the page number output of the *documentation navigation* model.

In SAL, this model composition was encoded as shown below.

```
 integrated_framework:  MODULE =
   documentation_navigation []
     (signifier || affordance || display_controlLogic || endUser_device_-
interaction ||
      plant || constraints) []
   task;
```

9.11 Specifications

One of each specification described in Section 9.2.1 (10 in total) was instantiated to evaluate applicability of the integrated framework approach. Seven specifications were encoded to evaluate the interface with respect to the modified pump stopped alarm troubleshooting procedure:

1. Three accuracy-related specifications involve understandability, completeness, and time efficiency while the pump stopped alarm procedure is executing

2. Four error tolerance-related specifications (including *accuracy and error tolerance*) involve the end user's opportunity to reconnect a disconnected lithium-ion battery to the replacement controller

3 specifications involve understandability, time efficiency, or completeness with respect to either procedure (pump speed adjustment or pump stopped alarm troubleshooting). Each specification is listed and explained below, and SAL syntax is provided in Appendices H.3.4–H.3.6.

1. *Accuracy and understandability*

As mentioned, the modified controller was designed to address the *visual consistency* signifier specification violation uncovered in Chapter 7. This modification was implemented to improve understandability; however, it is also critical that what is signified reflects the device's operational state (i.e., signifiers should be accurate). Because the pump stopped alarm has visual and audible properties, redundancy is also critical to understandability (i.e., what is signified by audible and visual properties of the alarm should not conflict).

During the pump stopped alarm troubleshooting procedure, these characteristics are needed to support the end user in identifying whether or not the alarm has been resolved. Thus, the *accuracy*

and understandability specification reads, "it is always true that if the device is operating within data-constrained parameters, the pump stopped alarm troubleshooting procedure is executing, and the pump stopped alarm is engaged, this implies that a pump speed of "Stopped" is signified consistently through the visual channel and redundantly through the audible/visual channels. The SAL syntax of this specification is provided in Appendix H.3.1.

2. *Accuracy and error tolerance*

One accuracy-related usability problem identified in Chapter 5 involved the end user accidentally connecting a previously disconnected lithium-ion battery to the replacement controller during the pump stopped alarm troubleshooting procedure. This situation was considered unsafe because the battery that was in-use when the alarm engaged could have been discharged or malfunctioning, and erroneously reconnecting it to the replacement controller cannot resolve the alarm. One way such a situation could be prevented is if an affordance enabling the end user to connect a previously disconnected lithium-ion battery to the replacement controller does not emerge if the modified procedure is executed as-written. An *accuracy and error tolerance* specification was encoded to ensure this characteristic of the modified interface. It reads, "it is always true that if the device is operating within data-constrained parameters, the pump stopped alarm troubleshooting procedure is executing, and either:

- the old lithium-ion battery cable is connected to the old lithium-ion battery, or
- the old lithium-ion battery cable is connected to the old controller

(i.e., it was in-use when the alarm engaged), this implies that the old lithium-ion battery cable is not connectable to the new controller and the new lithium-ion battery cable is not connectable to the old lithium-ion battery." The SAL syntax of this specification is provided in Appendix H.3.2.

3. *Accuracy and time efficiency*

The *accuracy and error-tolerance* specification could help ensure that a discharged or malfunctioning battery cannot be connected to the replacement controller. It is also critical that a fully charged

replacement battery can be connected quickly (i.e., as soon as the end user knows the replacement battery is fully charged). An *accuracy and time efficiency* specification was encoded to ensure that the interface supports such a behavior. It reads, "it is always true that if the device is operating within data-constrained parameters, the pump stopped alarm troubleshooting procedure is executing, the new lithium-ion battery has no lights illuminated, and the new lithium-ion battery has five lights illuminated in the next-state, this implies that affordances enabling the end user to connect the new lithium-ion battery to the new controller emerge in the next-state." The SAL syntax of this specification is provided in Appendix H.3.3.

4. *Accuracy and completeness*

In support completeness, it is critical that a pump speed is signified through at least one channel; and in support of accuracy, a pump speed of "Stopped" should be signified when the pump stopped alarm is engaged and the pump stopped alarm procedure is executing. Both of these characteristics are needed during the procedure for the end user to know whether the alarm has been resolved. Thus, the *accuracy and completeness* specification reads, "it is always true that if the device is operating within data-constrained parameters, the pump stopped alarm troubleshooting procedure is executing, and the pump stopped alarm is engaged, this implies that a pump speed of "Stopped" is signified through the visual, audible, or documentation channels."

5. *Understandability and error tolerance*

This specification combines:

- Understandability with respect to consistency of audible/visual properties on the device signifying pump speed and power supplied
- Error tolerance with respect to connecting a potentially discharged or malfunctioning battery to the replacement controller

Accuracy-related specifications assert either of these characteristics while the pump stopped alarm procedure is executing; however, it is also critical that understandability is satisfied in other sit-

uations, such as after the pump stopped alarm procedure has completed executing. Thus, the *understandability and error-tolerance specification* reads, “it is always true that:

- an affordance enabling the end user to connect a discharged or malfunctioning battery to the replacement controller never emerge; and,
- it is always true that pump speed and power supplied are signified consistently through the audible and visual channels”

The SAL syntax of this specification is provided in Appendix H.3.5.

6. *Understandability and time efficiency*

As mentioned, one alarm at a time can be engaged on the case study system’s controller. In the model, it is possible for the pump stopped alarm to disengage during the pump stopped alarm troubleshooting procedure; and if the pump stopped alarm disengages, the high power alarm could engage. In such a situation, it is critical that signifiers of pump speed and power supplied are understandable to the end user. Thus, the *understandability and time efficiency* specification reads, “it is always true that if the device is operating within data-constrained parameters, the pump stopped alarm is engaged, and the high power alarm engages in the next-state, this implies that power supplied and pump speed are signified consistently through audible/visual channels and power supplied is signified redundantly through audible/visual channels in the next-state.” The SAL syntax of this specification is provided in Appendix H.3.6.

7. *Understandability and completeness*

This specification combines:

- Understandability with respect to pump speed and power supplied signified through audible/visual channels
- Completeness with respect to pump speed and power supplied signified through audible, visual, and documentation channels

The *accuracy and understandability* and *accuracy and completeness* specifications only considered what is signified while the pump stopped alarm troubleshooting procedure is executing and the pump stopped alarm is engaged. In the other two understandability-related specifications, audible/visual signifier consistency could be satisfied by virtue of the unsafe situation in which they all signify nothing (i.e., understandability is satisfied, but completeness is violated). Thus, the *understandability and completeness* specification is needed to ensure that the interface is both understandable and complete in all continuous states matching simulation data and all discrete states, including those that emerge before, during, and after execution of all procedural steps. It reads, “when the device is operating within data-constrained parameters, this implies that:

- it is never true that neither pump speed nor power supplied is signified through at least one channel (audible, visual, or documentation); and,
- it is always true that pump speed and power supplied are signified consistently

The SAL syntax of this specification is provided in Appendix H.3.9.

8. *Error tolerance and time efficiency*

As mentioned in Section 9.2.1, a potential usability problem identified in Chapter 5 involved:

- The end user connecting a new lithium-ion battery to the new controller without checking its charge level first
- The end user being able to reconnect a potentially discharged or malfunction lithium-ion battery that was disconnected earlier

In support of error tolerance, the redesigned interface should prevent the end user from reconnecting a previously disconnected lithium-ion battery; and in support of time efficiency, the charge level of a new lithium-ion battery should be checked before it is connectable to the replacement controller. Thus, the *error tolerance and time-efficiency* specification reads, “it is always true that when the old lithium-ion battery cable is connected to the the old controller (i.e., the old lithium-ion battery was in-use when the pump stopped alarm engaged) and the new lithium-ion battery has 0 lights

illuminated. In the next-state, if the new lithium-ion battery has five lights illuminated (i.e., it is fully charged), this implies that:

- the new lithium-ion battery cable is not connectable to the old lithium-ion battery
- the new lithium-ion battery cable is connectable to the new lithium-ion battery"

The SAL syntax of this specification is provided in Appendix H.3.7.

9. *Error tolerance and completeness*

This specification combines:

- Error tolerance with respect to connecting a potentially discharged or malfunctioning battery to the replacement controller
- Completeness with respect to pump speed and power supplied signified through audible, visual, and documentation channels

Specifications encoded thus far have asserted either of these characteristics. The *error tolerance and completeness* specification is needed to ensure that both are satisfied concurrently. It reads, "it is always true that:

- affordances enabling the end user to connect a discharged or malfunctioning battery to the replacement controller never emerge; and,
- when the device is operating within data-constrained parameters, this implies it is never true that a pump speed or power supplied is not signified through audible, visual, or documentation channels"

The SAL syntax of this specification is provided in Appendix H.3.8.

10. *Time efficiency and completeness*

In regard to the pump speed adjustment procedure, the interface should support the end user in identifying whether progress has been made toward setting a desired speed. In support of completeness, pump speed should be signified through at least one channel; and in support of time-efficiency,

a pump speed should be signified immediately after a desired pump speed has been achieved. Thus, the *time-efficiency and completeness* specification reads, "it is always true that when the device is operating within data-constrained parameters, the speed setting is not equal to the desired speed, and the speed setting is equal to the desired speed in the next-state, a pump speed is signified through at least one channel in the next-state." The SAL syntax of this specification is provided in Appendix H.3.10.

9.12 Verification

Using the technique described in Section 9.2.2, each specification was verified by invoking SAL-INF-BMC at the default depth of 10. No counterexamples were returned. To compute the maximum depth of a possible counterexamples, *documentation navigation*, *end user-device interaction*, *affordance*, and *task* models were asynchronously composed within a discrete framework model. The specification encoded generally in (9.15) was instantiated for the discrete framework model (SAL syntax shown below) and verified using SAL-SMC.

```
G(NOT(aRespondToPumpStoppedAlarm_Executing AND aRespondToPumpStoppedAlarm_Done
      aAdjustSpeed_Executing AND aAdjustSpeed_Done);
```

SAL-SMC returned "proved" after 116 iterations. To ensure that no counterexamples exist in the integrated framework model, each specification was verified again by invoking SAL-INF-BMC at a depth of 116.

9.12.1 Results

Model checking results and verification times are reported in Table 9.2. The counterexample to *accuracy and understandability* returned a 52-step trace through the model leading up to an unsafe state in which:

- The pump stopped alarm troubleshooting procedure is executing
- The pump is operating within data-constrained parameters
- The pump stopped alarm is engaged

Table 9.2: Model checking results

Specification name	Result	Execution time (s)
Accuracy and understandability	*counterexample*	149,489.98
Accuracy and error-tolerance	*proved*	307,032.41
Accuracy and time-efficiency	*proved*	166,194.96
Accuracy and completeness	*proved*	70,945.93
Understandability and error-tolerance	*proved*	24,004.01
Understandability and time-efficiency	*proved*	21,803.0
Understandability and completeness	*proved*	189,181.66
Error-tolerance and time-efficiency	*proved*	24,705.58
Error-tolerance and completeness	*proved*	5,720.02
Completeness and time-efficiency	*proved*	22,962.96

- Visual channel properties on the device signify a pump speed of "Stopped" consistently
- Audible channel properties do not signify a pump speed

This counterexample is visualized in Figs. 9.28–9.35. Each figure depicts states of interest through the counterexample trace for *plant*, *documentation navigation*, *task*, *end user-device interaction*, *affordance*, and *signifier* models. The trace through the plant model is shown in Fig. 9.28. In the initial state, power, speed, and flow are 0. These values do not change in subsequent steps.

The trace through the documentation navigation model is shown in Fig. 9.29. Because an initial state of $iPage = 2$ was assigned, the trace represents the end user starting on page-2 (the table of contents); and in the next-state, the end user navigates to page-13 containing steps 1–4 of the pump stopped alarm troubleshooting instructions. Finally, the end user remains on page-13 for remaining steps of the trace.

The trace through the *task* model is shown in Fig. 9.30. This trace represents the end user executing step 1a, 2a, and 2b of the pump stopped alarm troubleshooting procedure in order. The ordering of actions is listed in outline form below, including the corresponding affordances that are actualized (trace through the *affordance* model discussed next):

1. Attempting to fix the connector permanently attached to the heart, which has broken and

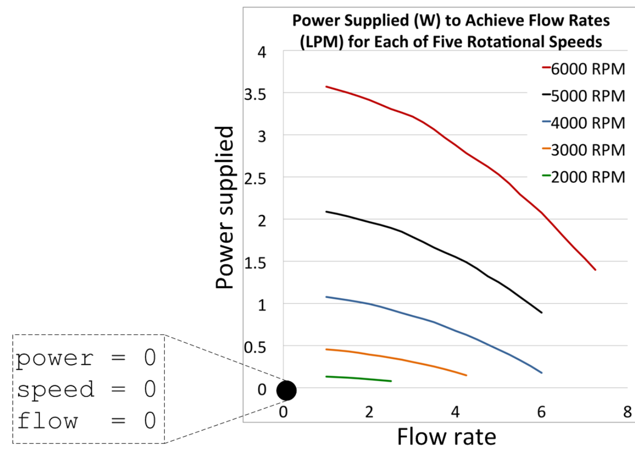


Figure 9.28: Graphical representation of the case study counterexample trace through the *plant* model. Flow rate in liters per minute (LPM) is shown on the x-axis. Power supplied in watts in shown on the y-axis. Colored lines shown the flow rate and power supplied for each speed setting 1–5 (corresponding to 2,000—6,000 RPM). The black circle indicates that the initial and final states of power, speed, and flow, are 0 (variables and values listed inside dashed-box rectangle)

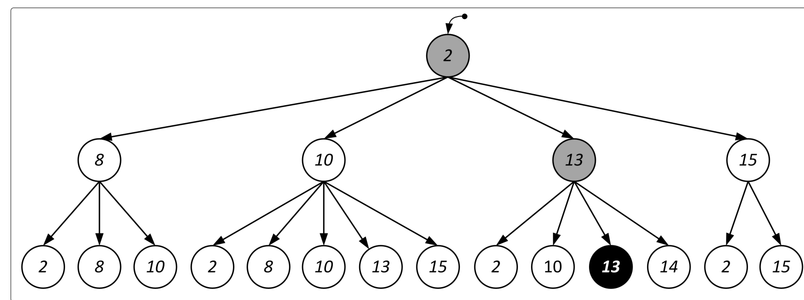


Figure 9.29: Graphical representation of the case study counterexample trace through the *documentation navigation* model. The curved arrow indicates that the initial state (step-0) is page-2. Straight arrows indicate next-states that were possible in the counterexample trace. Grey circles indicate the pages are along the path. The black circle indicates that the final state is page-13; i.e., *iPage = 13* for all remaining steps of the counterexample trace

come apart, by:

(a) Reassembling the connector parts (occurs for the first time at step-3 of the trace) by executing the action *hReassembleBrokenConnector* and actualizing the affordance *ConnectorPartsAssemblable*

(b) Performing the following steps three times (represented over steps 7–37 of the trace):

i. Disassembling the parts by executing the action *hDisassembleConnector* and actualizing the affordance *ConnectorPartsDisassemblable*

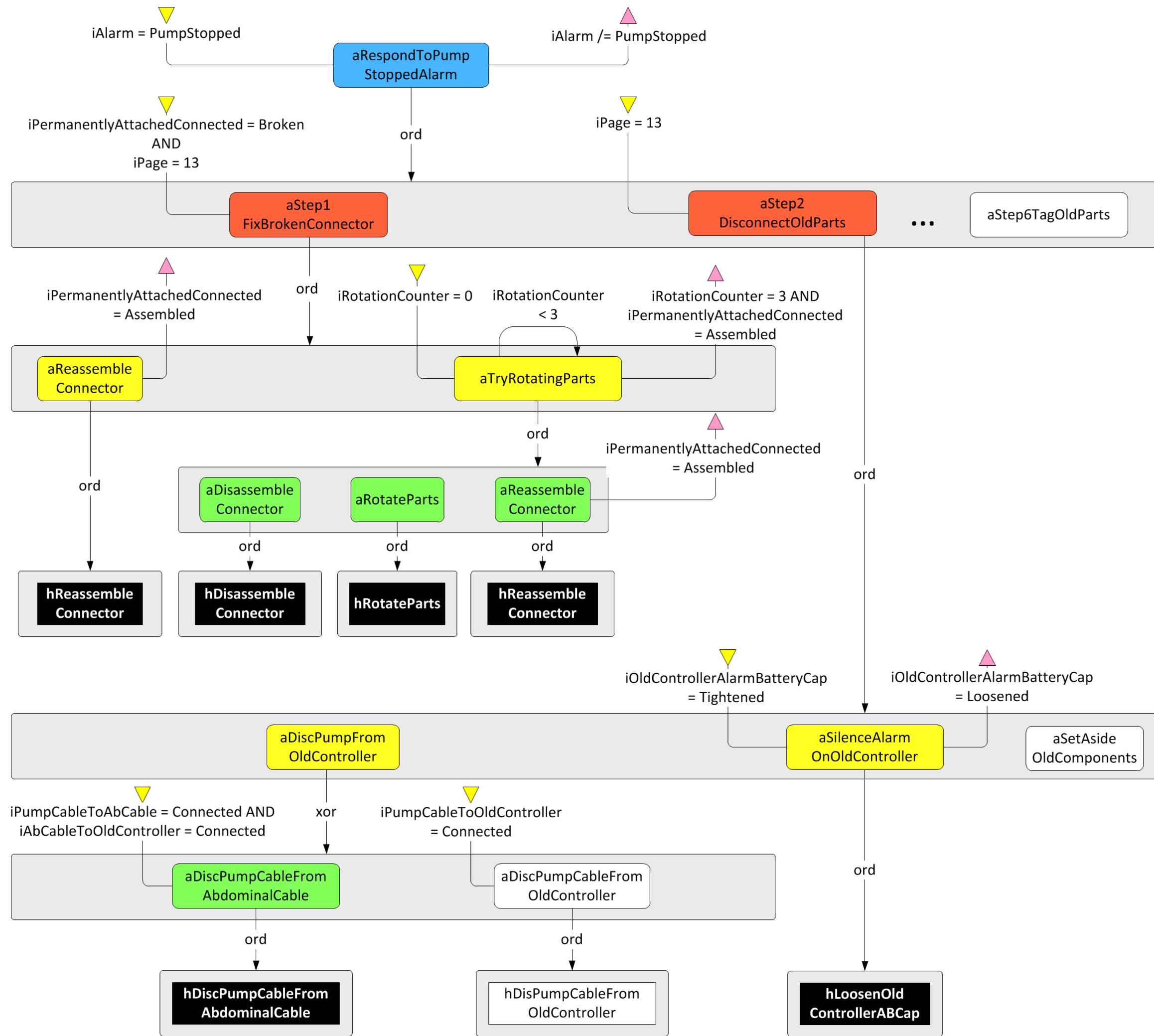


Figure 9.30: Graphical representation of the case study counterexample trace through the pump stopped alarm troubleshooting procedure *task* model. White, rounded-edge rectangles are activities that did not execute. White, square-edge rectangles are human actions that did not execute. Colored, rounded-edge rectangles are activities that executed. Rectangles having the same color are heterarchical. Black, square-edge rectangles are human actions that executed

ii. Rotating the parts 90° by executing the action *hRotateConnectorParts* and actualizing the affordance *ConnectorPartsRotatable*

iii. Reassembling the parts by executing the action *hReassembleBrokenConnector* and actualizing the affordance *ConnectorPartsAssemblable*

2. Disconnecting parts that were in-use when the pump stopped alarm engaged by:

(a) Disconnecting the abdominal cable from the old controller by executing the action

hDiscPumpCableFromAbCable and actualizing the affordance *PumpCableDisconnectable-FromAbCable* (occurs at step-45 of the trace)

(b) Silencing the alarm on the old controller by executing the action *hLoosenOldController-ABCap* (no corresponding affordance specified, occurs at step-51 of the trace)

The trace through *end user-device interaction* and *affordance* models are shown together in Figs. 9.31–9.33. In the initial states (Fig. 9.31):

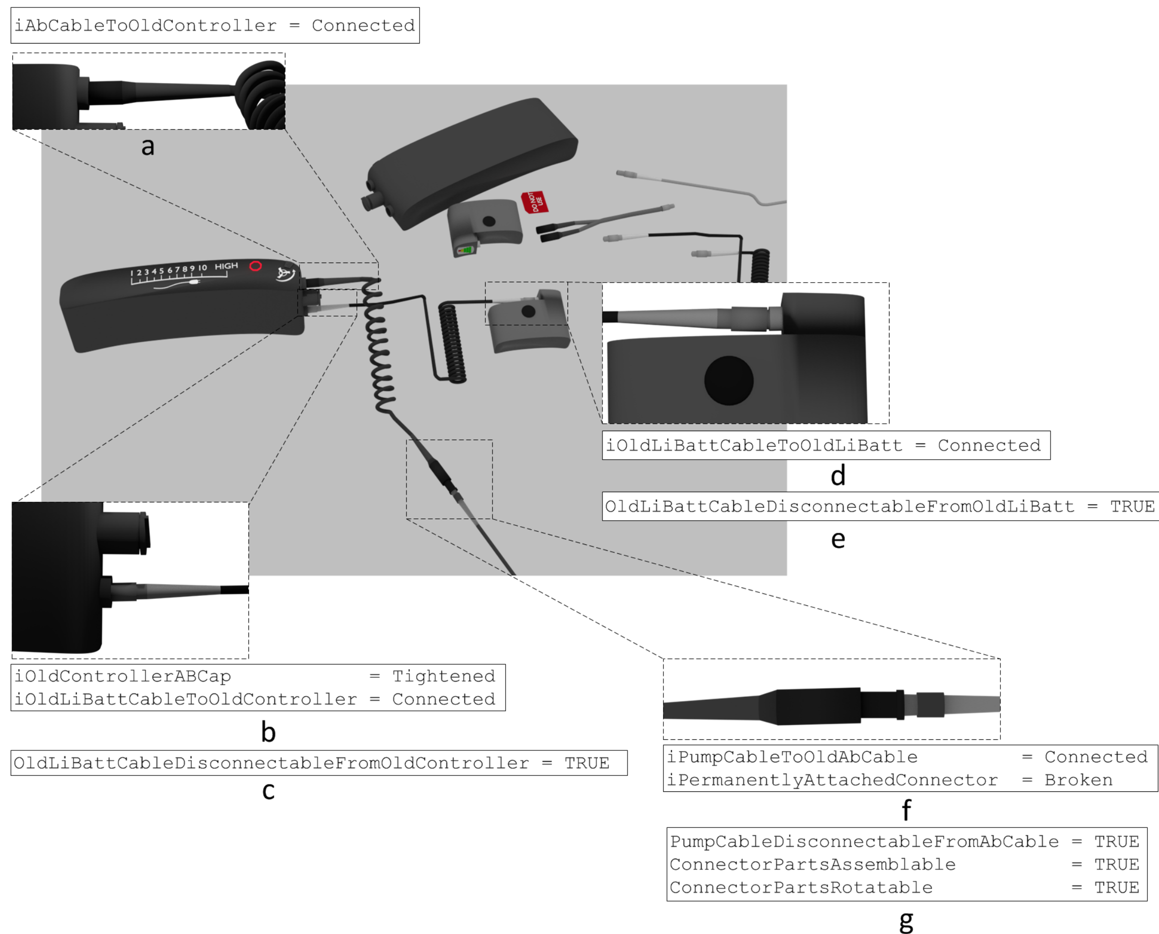


Figure 9.31: Graphical representation of *end user-device interaction* and *affordance* model initial states in the case study counterexample. Letters are added for reference in text of Section 9.12.1. In a–d, dashed-line rectangles contain enlarged graphical renderings of initial states for a subset of configurable hardware components. Solid-line rectangles show the corresponding variable value(s) from the end user-device interaction model. In e–g, solid-line rectangles show variable of interest from the *affordance* model)

- The abdominal cable is connected to the old controller (Fig. 9.31a)

- The old lithium-ion battery cable is connected to the old controller (Fig. 9.31b) and the end user can disconnect it (Fig. 9.31e)

- The old lithium-ion battery cable is connected to the old lithium-ion battery (Fig. 9.31c) and the end user can disconnect it (Fig. 9.31f)

- The pump cable is connected to the abdominal cable (Fig. 9.31d, first variable) and the end user can disconnect it (Fig. 9.31g, first variable)

- The connector permanently attached to the heart has broken and come apart (Fig. 9.31d, second variable) and the end user can reassemble it (Fig. 9.31g, second variable) as well as rotate the parts 90° (Fig. 9.31g, third variable)

The first changes to the *end user-device interaction* and *affordance* models occur at step-4 of the counterexample trace (Fig. 9.32). The end user executes the action prescribed in step 1a of the troubleshooting procedure, reassembling the connector permanently attached to the heart by pushing its two parts together. Executing this action changes the *end user-device interaction* model variable *iPermanentlyAttachedConnector* from *Broken* to *Assembled* (Fig. 9.32a). In the *affordance* model, assembling the connector renders the end user's opportunity to assemble it and rotate its parts unavailable; while the opportunity to dissemble the connector emerges (Fig. 9.32b). The first time these states occur, the end user-device interaction model variable *iRotationCounter* is 0. In subsequent states, it increases as the task model represents the end user executing step 1(a)ii of the troubleshooting procedure.

Over steps 5–50 of the counterexample trace, the *end user-device interaction* and *affordance* models transition between states represented in Figs. 9.31 and 9.32. This is because, over these steps, the task model is representing the end user repeatedly disassembling, rotating, and assembling parts of the permanently attached connector. As mentioned, at step-51, the task model represents the end user silencing the alarm on the old controller by loosening the 1/2 cell AA battery cap. This leads to the final states of *end user-device interaction* and *affordance* models at step-52 (Fig. 9.33). When the end user loosens the alarm battery cap on the old controller, the *end user-device interaction*

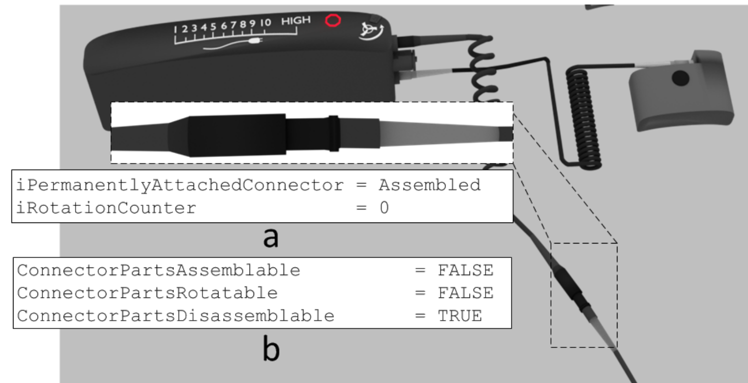


Figure 9.32: Graphical representation of *end user-device interaction* and *affordance* model states in intermediate steps of the case study counterexample. Letters are added for reference in text of Section 9.12.1. The dashed-line rectangle contains an enlarged graphical rendering of the connector permanently attached to the hear. In (a), the solid-line rectangle shows the corresponding variable values from the *end user-device interaction* model. In (b), the solid-line rectangle shows the corresponding variable values from the *affordance* model

model variable *iOldControllerABCap* transitions from *Tightened* to *Loosened* (Fig. 9.33a). The initial state of *iNewControllerABCap* (which is also the final state) is also shown in Fig. 9.33b, since it is relevant to the unsafe state.

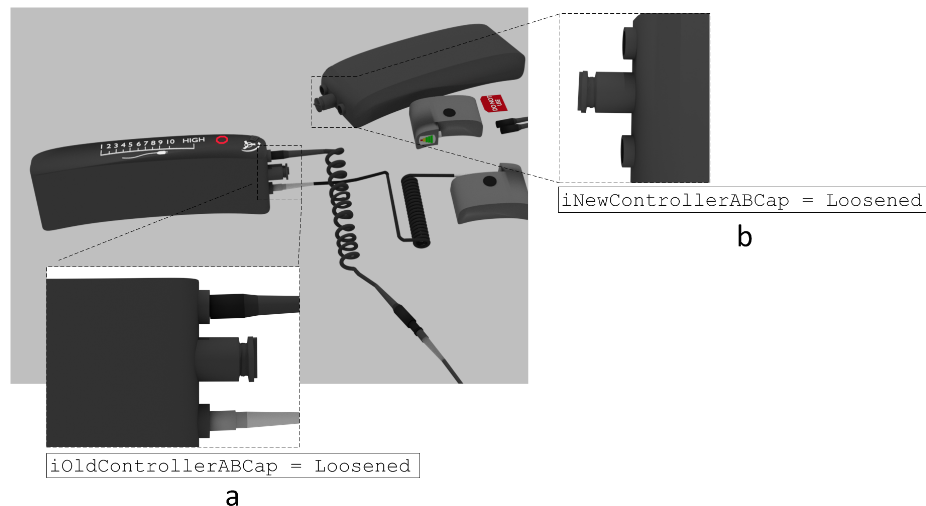


Figure 9.33: Graphical representation of *end user-device interaction* and *affordance* model final states in the case study counterexample. Letters are added for reference in text of Section 9.12.1. Dashed-line rectangles show enlarged graphical renderings of final states for the old and new controllers. Solid-line rectangle shows corresponding variable values from the end *user-device interaction* model

The trace through *signifier* and *display/control logic* models are shown together in Figs. 9.34 and 9.35. The value of *functional* is not shown; however, it is *false* in every state of the counterexample

trace. In the initial state of the *display/control logic* model, the pump stopped alarm is engaged (Fig. 9.34a) and no power indicator lights are illuminated (Fig. 9.34b). These states do not change in steps 1–52 of the counterexample trace. In the initial state of the signifier model, the volume (Fig. 9.34c), audible pattern (Fig. 9.34d), and color (Fig. 9.34e) of the pump stopped alarm signify a pump speed of “Stopped.” The pump speed signified by the speed setting knob label depends on the color of the pump stopped alarm light; thus, it signifies a pump speed of “Stopped” (Fig. 9.34f). These states do not change in steps 1–51 of the counterexample trace.

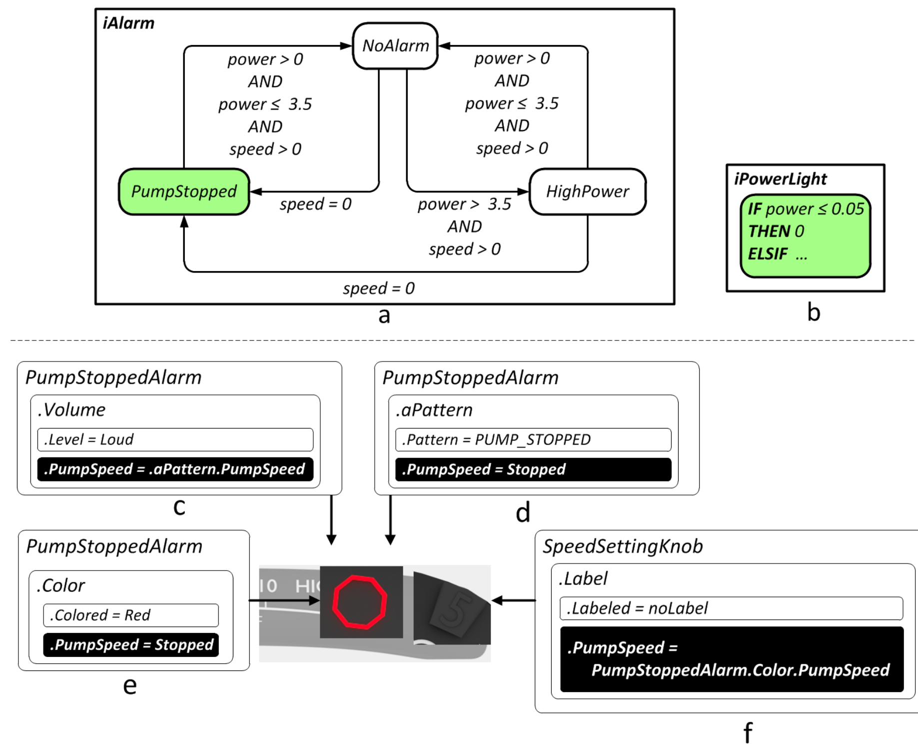


Figure 9.34: Graphical representation of *signifier* and *display/control logic* model initial states in the case study counterexample. Letters are added for reference in text of Section 9.12.1. In (a) and (b), rounded-edge rectangles highlighted green indicate what initial states were assigned in the *display/control logic* model

In the final state of the *signifier* model (step-52 of the counterexample trace), the volume (Fig. 9.34a) and audible pattern (Fig. 9.34b) of the pump stopped alarm do not signify pump speed. This situation emerged because the 1/2 AA alarm battery caps on both controllers are loosened, and no audible alarms are engaged (i.e., there are no identifiable audible patterns or volumes for the end user). Color of the pump stopped alarm (Fig. 9.34c) and label of the speed setting (Fig. 9.34d)

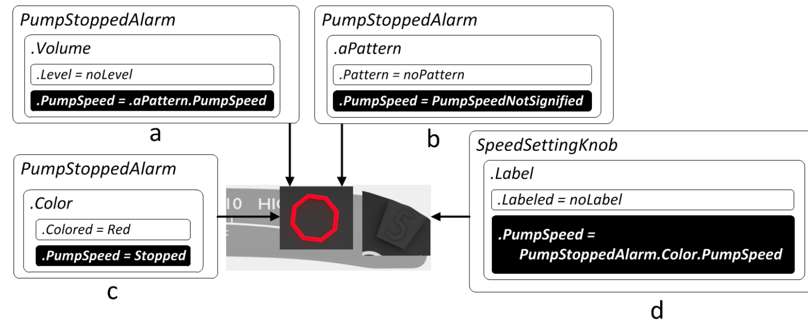


Figure 9.35: Graphical representation of *signifier* model final states in the case study counterexample. Letters are added for reference in text of Section 9.12.1

signify a pump speed of "Stopped." Here, visual and audible channel signifiers are both internally consistent; however, conflicting meanings are signified through the audible channel.

Verification reports for the other three accuracy-related specifications returned no counterexamples. These results indicate that executing the pump stopped alarm troubleshooting procedure as-written ensures that:

1. The end user can never connect the old lithium-ion battery to the new controller (i.e., no counterexamples were returned for *accuracy and error tolerance*)

2. A new, fully charged lithium-ion battery is always connectable to the new controller immediately after the end user checks its charge level (i.e., no counterexamples were returned for *accuracy and time efficiency*)

3. A pump speed of "Stopped" is always signified through at least one channel (i.e., no counterexamples were returned for *accuracy and completeness*)

Verification reports for the remaining six specifications also returned no counterexamples. These results indicate that:

1. If a lithium-ion battery is in-use when the pump stopped alarm engages, the end user can never connect the old lithium-ion battery to the new controller; additionally, visual/audible signifiers of pump speed and power supplied are always consistent (i.e., no counterexamples were returned for *understandability and error tolerance*)

2. If the alarm transitions from pump stopped to high power, visual/audible signifiers of pump speed and power supplied are always consistent and redundant in the next-state (i.e., no counterexamples were returned for *understandability and time-efficiency*)
3. A pump speed and a power supplied are always signified through at least one channel; additionally, visual/audible signifiers of pump speed and power supplied are always consistent (i.e., no counterexamples were returned for *understandability and completeness*)
4. If a lithium-ion battery is in-use when the pump stopped alarm engages and the end user checks the charge level of the new lithium-ion battery, in the next-state, the end user can connect the new lithium-ion battery cable to the new lithium-ion battery but cannot connect it to the old lithium-ion battery (i.e., no counterexamples were returned for *error tolerance and time efficiency*)
5. If a lithium-ion battery is in-use when the pump stopped alarm engages, the end user can never connect the new lithium-ion battery cable to the old lithium-ion battery; additionally a pump speed and a power power supplied are always signified through at least one channel (i.e., no counterexamples were returned for *error tolerance and completeness*)
6. If the end user adjusts the speed setting, in the next-state, a pump speed and a power supplied are always signified through at least one channel

9.13 Discussion

This chapter has introduced an integrated framework for the formal modeling and verification of human-interactive system usability. Leveraging the modeling methodologies developed in this research, an integrated model architecture was developed to support the analyst in representing a broad range of interactions. To support verification, usability-related measures of accuracy, understandability, error tolerance, time efficiency, and completeness were leveraged to derive ten specifications and their LTL encodings. Symbolic and infinite-bounded model checkers were leveraged to verify specifications in a way that considers all discrete states and the subset of continuous states matching simulation data.

One implementation of the framework was demonstrated in a case study based on a pediatric blood pump under development and a modified version of an existing system's interface. Modifications were informed by safety-critical system usability measures and design insights gleaned in prior chapters. Formal models were encoded and composed within a system model to represent interactions among:

- Navigation through a 29-page printed document and:
  - Signifiers operating through the documentation channel
  - One operational and one troubleshooting procedure provided in text
- The two procedures in documentation and end user-device interaction
- End user-device interaction and:
  - 13 affordances, 11 of which have corresponding motor actions prescribed in the troubleshooting procedure
  - A constrained set of spatial relations among cables and connectors
  - The target system's control logic
- The target system's control logic and:
  - Signified pump speed and power supplied (through visual, audible, and documentation channels)
  - Actual pump speed and power supplied

One of each LTL usability specification was instantiated and verified using infinite-bounded model checking. Model checking results indicate that a problem related to accuracy and understandability could emerge for the end user while executing the pump stopped alarm troubleshooting procedure: while the audible alarms of both controllers are temporarily silenced, conflicting pump speeds are signified through the audible and visual channels.

Steps leading up to this unsafe state show that the interface has some desired characteristics:

- The trace through the *documentation navigation* model shows that the end user can locate the pump stopped alarm troubleshooting procedure in one-step from the table of contents
- The trace through the *task* model shows that the modified procedure may reflect a time-efficiency improvement with respect to the original procedure: the end user attempts to fix the connector permanently attached to the heart (a corrective action identified in Chapter 5) before attaching red tags to old components (a non-corrective action identified in Chapter 5)
- The traces through *end user-device interaction* and *affordance* models show that affordances support the end user in completing procedural steps of assembling, disassembling, and rotating parts of the permanently attached connector as well as disconnecting the pump cable from the abdominal cable
- The traces through *display/control logic* and *signifier* models show that visual/audible signifiers of pump speed are always internally consistent in the scenario represented in the counterexample
- Considering states of the *plant* model that are constant throughout the counterexample trace, pump speed signified through the visual channel is accurate with respect to the actual pump speed

No counterexamples were returned for the other nine specifications, indicating that the human-system interface can be considered usable with respect to three accuracy-related specifications and six specifications involving intersecting aspects of understandability, time efficiency, and error tolerance.

9.13.1 Methodological Considerations

The integrated model architecture is the first to consider documentation, configurable hardware, displays, controls, and actuators as interacting elements of a human-integrated system. The architecture was designed to support the analyst in combining the modeling methodologies developed in this research and by other researchers. This was accomplished by allowing different combinations of input/output variables to be exchanged between models. Specifications considered only two intersecting measures of safety-critical system usability and only one interpretation of what interface

properties satisfy these measures. Case study results indicate that at least one implementation of the framework and one set of usability measure interpretations could be useful early in the design cycle of a medical device.

While it is possible that the case study counterexample to *accuracy and understandability* reflects a usability problem, it is unclear if temporarily silencing the audible alarms of both controllers will be problematic for an actual end user. As mentioned in Section 9.9.2.1, a situation in which both controllers emit audible alarms could also be problematic; thus, modifying the pump stopped alarm troubleshooting procedure so that both 1/2 cell battery caps are tightened may not resolve the problem. This issue could be explored further in a validation study involving human participants.

Some of the tools and techniques from prior chapters, including the EOFM-to-SAL translator developed by other researchers [10], helped facilitate the model development process. However, some SAL model infrastructure needed to be encoded manually. The *display/control logic*, *end user-device interaction*, *HES*, and *plant* models needed to be developed without tool support; while the SAL syntax of automatically generated *signifier* and *constraints* models needed to be modified. Understandability-related specifications were partially supported by the automated specification generation tool from Chapter 7; however, they needed to be modified manually, and all other specifications were developed without tool support.

Chapter 10: Contributions and Future Work

This research has provided new and significant contributions to formal modeling and verification methodologies, which enable a broad range of analyses for human factors engineers (Table 10.1). Formal modeling, specification, and verification were leveraged in different ways to identify a new

Table 10.1: Human-system interface elements that must be modeled and usability measures that must be verified in safety-critical systems. "✓ " denotes that an area of methodological support has been addressed by other researchers; "★" denotes that an area was addressed this work; and '—" denotes and area must be addressed in future work

Usability measure	Interface element	Methodological support for the analyst					
		Developing formal models			Conducting model checking		
		Formalism	Technique	Tool	Spec.	Technique	Tool
Accuracy	Displays	✓	✓	✓	✓	✓	✓
	Control logic	✓	✓	✓	✓	✓	✓
	Hardware	★	★	★	★	★	★
	Documentation	✓	★	✓	—	—	—
Understand-ability	Displays	★	★	★	★	★	★
	Control logic	✓	✓	✓	✓	✓	✓
	Hardware	★	★	★	★	★	★
	Documentation	✓	★	✓	★	★	—
Error tolerance	Displays	—	—	—	—	—	—
	Control logic	✓	✓	✓	✓	✓	✓
	Hardware	✓	★	★	★	★	—
	Documentation	—	—	—	—	—	—
Time efficiency	Displays	✓	✓	✓	✓	✓	✓
	Control logic	✓	✓	✓	✓	✓	✓
	Hardware	—	—	—	—	—	—
	Documentation	✓	★	✓	★	★	—
Completeness	Displays	★	★	★	★	★	★
	Control logic	★	★	★	✓	★	★
	Hardware	—	—	—	—	—	—
	Documentation	✓	★	✓	—	—	—

set of usability problems that could emerge for the end user while interacting with documentation,

displays, controls, and configurable hardware. Existing frameworks that are applicable to displays, controls, approximate actuator dynamics, and goal-driven human task behaviors were extended by integrating documentation, configurable hardware affordances, signifiers, and precise actuator dynamics as elements of the human-interactive system. This was accomplished by developing novel tools and techniques, applying them within a series of medical device case studies, and combining them within an integrated framework that considers intersecting usability measures with respect to a broad range of modeled interactions.

Chapters 4–8 built toward the framework by supporting the formal modeling of documentation, signifiers, affordances, and controlled actuators. New formalisms were developed to characterize documentation navigation and signifiers. To instantiate the documentation navigation formalism, a modeling technique was developed using the native syntax of SAL [68]. To instantiate the signifier formalism, a modeling technique and encoding tool were developed to support the analyst in specifying what is signified using a custom, XML-based language, an automated translation tool, and an additional technique for incorporating a model of the device. Leveraging existing formalisms, new modeling techniques and encoding tools were developed to enable the formal modeling of procedures in documentation, Gibsonian affordance, and controlled actuators. This research extended the modeling capabilities of existing frameworks in three ways:

1. By considering documentation as part of the human-system interface, the analyst can model documentation navigation, procedures as-written, and interactions between content in documentation and electrical/mechanical device components
2. By encoding signifier and affordance models, the analyst can model interactions between the target system and a constrained set of end-user characteristics
3. By replicating simulation data within model checking analyses, the analyst can represent controlled actuator behaviors without the need for differential equation abstraction

To verify safety-critical system usability, researchers have developed temporal logic specifications, model checking techniques, and encoding tools. These methodologies have mainly focused on veri-

fying accuracy, understandability, error tolerance, time efficiency, and completeness of displays and control logic (Table 10.1, denoted by check marks last three columns). The LTL specifications and model checking techniques developed in this research (Table 10.1, denoted by stars in the last three columns) extend the capabilities of existing methodologies with respect to:

- Understandability of the interface with respect to signifiers
- Time efficiency of documentation with respect to:
  - Page reachability
  - The ordering of preparatory and corrective actions in procedures
- Accuracy and error-tolerance of configurable hardware
- Completeness of the interface with respect to signifiers on the device and explanations in accompanying documentation

For each modeling and verification methodology, medical device case studies were utilized to demonstrate an application. Results indicated the model-based methodologies developed in this research can uncover potential usability problems in existing systems using highly abstracted models and automated verification.

In regard to documentation navigation, case study results showed that page reachability problems could emerge for the end user if PDF hyperlinking functionality does not support navigation between a page containing procedural steps and a page containing information that is needed to complete them. The analysis was conducted without representing documentation content or low-level user tasks (e.g. keystroke level modeling [80]).

The case study analysis of procedures in documentation showed that formal modeling and verification processes could be independently useful. The task modeling techniques aided in identifying a potential accuracy-related problems regarding what device components are identified: a cable having two output ends was ambiguously described in text of the procedure, and the formal task model enabled the end user to disconnect it in a way that enables a discharged or malfunctioning batter

to be connected later in the procedure. The device modeling technique proved useful for identifying potential completeness-related problems involving what initial device configurations are possible vis-a-vis what configurations are addressed in the procedure: eight initial cable configurations were possible, but the instructions were only applicable to two. Model checking results showed that it is possible to identify potential time-efficiency improvements by enabling one or more main steps of the procedure to execute in any order. This was accomplished by instantiating two time-efficiency specifications regarding the ordering of actions and modifying one EOFM decomposition operator [10] in the formal task model.

In regard to configurable hardware, the CAVEMEN approach case study showed that:

- End-user opportunities to physically manipulate cable inputs and outputs can be modeled by instantiating an existing affordance formalism
- Evolving spatial relations among HES entities can be modeled based on what affordances are actualized
- Accuracy and error tolerance of configurable hardware can be verified using LTL and model checking

This was accomplished by selecting a medical device adverse event report from the MAUDE database [83] and analyzing it using the CAVEMEN approach. While it was necessary to encode HES model infrastructure and LTL specifications manually in SAL, CAVEMEN-XML proved useful for instantiating Greeno's formalism [6] in a mathematical way. Model checking results revealed that initial states of the HES supported a surgeon in connecting the leads of an implanted maker to correct pulse generator input ports; however, the same initial conditions also enabled the erroneous affordance of LV lead connectability to the RV port. Such a situation emerged in the adverse event, and one potential manifestation of the event was captured in a counterexample trace through the model. The scalability evaluation showed that the CAVEMEN approach is scalable using symbolic model checking on the target workstation in models representing up to 64 unique affordances (for comparison, the case study had three unique affordances).

The BIGSIS approach case study showed that formal methods can be utilized to represent signifiers formally based on a constrained set of visual, audible, and haptic properties on the device; explanations of what is signified in accompanying documentation; and end-user characteristics that could shape what is signified. Using BIGSIS-XML and the JavaScript-based translator, it was possible to instantiate the BIGSIS formalism and LTL signifier specifications in the model checking syntax of SAL, without the need for encoding them manually. A minimal model of the device's control logic and end user-device interaction proved useful for controlling end-user descriptions and encoding constrained specifications. Model checking results helped identify potential usability problems involving:

- Internal consistency of visual signifiers
- Redundancy of signifiers operating through visual and audible channels
- Consistency between what is signified on the device and explanations in accompanying documentation (captured using a redundancy specification)

The scalability evaluation indicated that a formal signifier model incorporating up to 64 guarded transitions can be verified using symbolic model checking on the target workstation (for comparison, the case study model had five guarded transitions).

In regard to controlled actuators, the MATLAB-based tool and a case study using medical device simulation data showed that continuous actuator dynamics can be incorporated within model checking analyses without using differential equation abstractions. The was accomplished by automatically generating a *constraints* model and composing it with a manually encoded *plant* model (based on the technique in [46]). Model checking results showed that a Boolean variable output of the *constraints* model could be leveraged within LTL specifications, enabling the analyst to execute model checking analyses that are constrained to the set of continuous states matching spreadsheet data. Scalability evaluation results showed that the approach could be employed using infinite-bounded model checking on the target workstation for models based on a spreadsheet having up to 65,536 cells.

The tools and techniques represented in Table 10.1 were integrated in Chapter 9. Leveraging various input/output variable exchange mechanisms, an architecture was developed to support the analyst in representing a broad range of interactions among human-interactive system elements. Different implementations of the architecture support new and existing ways of modeling human-interactive system behaviors. Paired measures of accuracy, understandability, time-efficiency, error tolerance, and completeness were leveraged within ten usability specifications that are applicable to instantiated framework models. A model checking technique combined symbolic and infinite-bounded model checking to support formal verification analyses that consider all discrete states and a constrained set of continuous states. One implementation was demonstrated in a case study based on a pediatric blood pump under development. Instantiating the model architecture showed that it is possible to represent a target system's end user, documentation, configurable hardware, displays, control logic, and actuators within an integrated, human-system interface model. Model checking results showed that the framework could be employed early in the design cycle to ensure usability and identify potential problems with respect to a constrained set of modeled interactions and specifications. One counterexample revealed a potential usability problem involving interactions among a troubleshooting procedure in documentation, the target system's control logic, and visual/audible signifiers. No counterexamples were found for nine of ten specifications, indicating that design insights gleaned in Chapters 5 and 7 helped improve usability of the existing system's interface, and no additional modifications are needed to improve the interface with respect to the usability properties captured in these specifications.

10.1 Specific Contributions

While formal models are typically developed to enable verification; in this research, the process of developing formal models was leveraged to analyze accuracy and completeness of procedures in documentation and to identify signifier specifications that should be verified using model checking. Thus, modeling and verifying the target system proved useful when analytic outputs were utilized together.

To support the analyst in applying the methodologies, formalisms, modeling techniques, encoding

tools, temporal logic specifications, model checking techniques, and a specification generation tool were developed. In Chapters 4 and 7, new formalisms leveraged the Z specification language [153]. The accompanying modeling techniques employed the native syntax of SAL. In Chapters 6 and 7, encoding tools were developed using XML and JavaScript. Researchers have leveraged the parsing capabilities of Java to translate XML-based representations to a model checking syntax [10]. To employ such a tool, the analyst needs to install Java and run the translator from either the command line or a third-party integrated development environment (IDE). By using JavaScript, the analyst can run translators from a desktop browser without installing additional software.

10.2 Future Work

Future work should address two areas:

- Practical applications of the framework (Section 10.2.1)
- Future development of the framework (Section 10.2.2)

10.2.1 Practical Applications

In general, this research centers on addressing methodological knowledge gaps at the intersection of human factors and formal methods; however, there are practical applications that should be explored in future work, including:

- Informing improved usability standards for safety-critical systems, such as ISO/IEC-62366 for medical devices [17].
- Evaluating usability of high-risk, high-reward medical devices that are eligible for the U.S. FDA Expedited Access Pathway (EAP) [222]

10.2.1.1 Informing Improved Usability Standards

In the safety-critical system standards literature, such as ISO/IEC-62366, usability is defined with respect to one broadly categorized measure and one interface element, such as understandability of a graphical display [17]. In Chapters 4–7, temporal logic specifications reasoned about one usability measure and one interface element in a more specific way, such as internal consistency of signifiers

operating through the same sensory channel. And in Chapter 9, integrated framework specifications defined usability with respect to intersecting usability measures and multiple, interacting interface components, such as internal consistency of signifiers operating through the same sensory channel and error tolerance of configurable hardware. In future work, it would be beneficial to explore ways of integrating these specifications within the standards literature, which could support improved usability analyses without the need for formal methods.

10.2.1.2 Evaluating Usability of EAP Medical Devices

In April 2015, the U.S. FDA implemented the EAP: an expedited regulatory approval pathway for medical devices that address an urgent, unmet public health need [222]. In the EAP paradigm, one way of facilitating expedited access to eligible device—potentially the pediatric blood pump evaluated in Chapter 9—is by relegating usability evaluations to the post-market period [222]. Thus, while the Chapter 9 case study proved useful for demonstration, results also indicate that the framework shows promise toward evaluating interface usability early in the design cycle, potentially supporting the EAP paradigm without eliminating premarket usability analyses or delaying patient access. This potential application should be explored further in future work.

10.2.2 Future Development of the Framework

Future work for improving the framework's constituent methodologies were discussed in Chapters 4–8. Many of the same areas should be addressed to enhance the integrated framework, including:

- Extensions to the modeling and verification methodologies (Section 10.2.2.1)
- Case study implementations (Section 10.2.2.2)
- Tool support (Section 10.2.2.3)

10.2.2.1 Extensions to the Modeling and Verification Methodologies

The integrated framework could be extended to improve modeling and verification capabilities. In regard to modeling, the integrated framework architecture currently centers on interface design in-

dependently of end-user training. This limits applicability of the framework with respect to training needs that inform design. For example, if an interface feature requires intensive training for end users, designers may remove that feature [223]. Currently, the framework lacks methods and measures for evaluating the interface with respect to such training considerations. One way to address this in future work involves developing methods and measures for modeling, specifying, and verifying training needs within the integrated framework.

The framework architecture currently supports the analyst in applying one model checking framework (SAL [68]), one implementation of SAL syntax, and one set of underlying formalisms. In future work, alternative implementations should be tested by leveraging other formalisms (e.g. the CTT task analytic notation [66]), alternative SAL constructs (e.g. HybridSAL [144]), and various model checking frameworks (e.g. NuSMV [224]). By allowing the analyst to utilize a variety of tools and techniques, the framework could have improved interoperability with tools developed by other researchers (e.g. the IVY workbench, which utilizes CTT and NuSMV [59]).

Testing alternative implementations could also aid in improving scalability. Currently, the necessary search depth of SAL-INF-BMC is mostly increased due to the XML-to-SAL translation protocol of EOFM [10], which produces many guarded transitions, and consequently, an exponential increase in verification time that is proportional to bounded model checking search depth [225]. While researchers have developed ways of decreasing the number of guarded transitions in automatically generated EOFM-SAL models [156], it could be possible that a different task analytic formalism is advantageous in certain applications. It is also possible that *affordance*, *signifier*, and *plant* model translation protocols could be improved in ways that decrease state space and verification time.

In regard to verification scope, the framework could be extended by leveraging different interpretations and combinations of usability measures to derive new specifications. Different interpretations of the usability measures could support a more complete set of analyses, such as error-tolerance of signifiers, understandability of procedures in documentation, and completeness of affordances. New formulations of the ten specifications involving two intersecting usability measures could also be developed using different interpretations. For example, accuracy and understandability could be

interpreted with respect to what affordances are signified. It could also be beneficial to intersect three, four, or all five safety-critical system usability measures within more inclusive specifications.

10.2.2.2 Case Study Implementations

In the integrated framework case study, the *display/control logic* model was highly abstracted, two of four alarms were omitted, and battery charge levels were not modeled. The *signifier* model did not consider signifiers of configurable hardware or all specifiable signifiers on the controller. The CAVEMEN-XML model precluded the full set of affordances that were applicable to both procedures; and in the *HES* model, spatial relations were constrained to a subset of input–output cable connections, while end-user motor capabilities were assumed to support all of the modeled affordances. It could be beneficial to encode a more detailed representation of the target system and more inclusive *signifier/affordance* models in future implementations of the framework. Such a model could capture the set of affordances that are signified, and the affordances that are needed to execute all motor tasks, including interactions with accompanying documentation. Additionally, only one implementation of the framework architecture was tested, and future work should explore the possible combinations identified in Fig. 9.1. Testing different implementations of the framework could allow for comparing the applicability of different theories embodied in alternate implementations of the architecture; for example, the framework allows analysts to utilize either affordances, signifiers, or device states as task execution conditions. The same target system and specifications could be analyzed using each implementation, and model checking counterexamples (if any) can be compared to assess the trade-offs between each approach.

Because one purpose of the case study was to demonstrate an application of the integrated specifications, only one of each specification was instantiated, and each mainly reasoned about one troubleshooting procedure. Future work should explore the value of encoding and verifying a greater number of specifications, such as all possible versions of *accuracy and understandability* mentioned in Section 10.2.2.1. A broad range of systems should also be tested to assess the applicability of specifications to usability problems observed in the literature (such as those recorded in U.S. national databases [203]), while more complex systems should be modeled in greater detail to assess

scalability.

10.2.2.3 Tool Support

Finally, the web-based tools developed in this research could be enhanced to better facilitate the model and verification processes. As discussed in prior chapters, it could be beneficial for such a tool to incorporate a graphical interface and point-and-click functionalities that reduce the need for XML and SAL code. Researchers have also developed tools that automatically generate formal models of control logic from implemented software [63, 59, 226], and it could be beneficial to provide similar tools for this framework. Other tools, such as PVSio [140], allow the analyst to upload an image of a device controller and use it to edit model syntax representing displays and control logic. By combining automatic generation of *display/control logic* models with uploaded images of the interface, it could be possible to enable automated generation of *signifier* model infrastructure relating states of the device to visual appearances of displays and controls (i.e., end-user descriptions). Open-source 3-D modeling applications, such as Blender [227], allow analysts to implement and animate the physical construction of device hardware using the Python programming language [228]. To facilitate the development of *affordance* models, it could be beneficial to develop a tool that is capable of parsing Python code and automatically generating SAL syntax, including the part-whole compositions of components and the spatial relations among them depicted in static or animated Blender renderings.

While manually encoding one of each specification proved useful in the integrated framework case study, it would be beneficial to explore ways of automatically generating the full set of specifications that are relevant. One way of accomplishing involves developing a tool that parses instantiated models and generates specifications, similar to the tools described in [57] and Chapter 7. For each of the ten specifications developed in Chapter 9, such a tool should be capable of generating all possible versions, such as multiple *accuracy and understandability* specifications that are applicable to each procedure and relevant states of the device.

In regard to the model checking technique, the combined application of symbolic and infinite-bounded model checking enabled analyses that were exhaustive with respect to discrete states and a

constrained set of continuous states. Currently, the analyst needs to execute the step of computing a necessary search depth manually, and support should be developed in future work to automate this process. Additionally, the counterexamples traces through each framework model needed to be visualized manually. Researchers have developed macros that automate counterexample visualizations [73], and it could be beneficial to develop a similar macro for the framework. In addition to search depth computation and counterexample visualization, other parts of the verification process that could be automated include:

- *Witness generation* (i.e. traces through the models leading up to state satisfying the specification). In Chapter 4, witnesses were automatically generated by SAL-WMC [68]; however, no such capability is available for SAL-INF-BMC. Such a capability should be explored in future work, as it would support the analyst in validating the models by attempting to recreate a desired trace while interacting with the target system (as done in Chapter 4)
- *Design improvements.* While model checking counterexamples could be leveraged to inform design improvements, it would be beneficial to develop a tool that automatically suggests or implements modifications that prevent the specification violation. For example, researchers have leveraged machine learning to automatically generate target system control logic that supports normative end-user task behavior [229]. For the framework, a similar approach could be developed to automatically implement or suggest improved versions of signifiers, documentation, configurable hardware, and control logic that ensure no specifications are violated

While these tool extensions could benefit the analyst, it is currently unclear what extensions are necessary for the approach to be considered efficient. For example, if the analyst needs extensive training to employ the framework, then its application may be inefficient relative to other usability evaluation methods. Thus, future work should identify what knowledge and skills are needed for the analyst to apply the framework using currently available web-based tools, modeling techniques, and verification techniques. Such an evaluation could identify the minimal training an analyst needs to employ the framework as compared with other methods, such as heuristic evaluation [34].

Leveraging these data, tools and accompanying training curricula could be developed to improve relative efficiency of the framework.

10.3 Conclusion

By providing novel modeling and verification methodologies, this research has extended the analytic capabilities of model-based approaches at the intersection of human factors and formal methods. Formalisms, modeling techniques, specifications, and model checking techniques were developed to enable a new set of formal usability analyses, while tools were developed to facilitate their application. A series of medical device cases studies showed that each approach could be applied individually to support analyses of documentation, affordances, and signifiers for existing systems. The integrated framework combined each approach and enabled holistic analyses of a prototype system under development, without the need for a complete implementation of its interface, a manual search for problems, or an interdisciplinary team of experts. These contributions support rigorous, iterative, and highly automated usability analyses that are applicable throughout the design cycle of safety-critical systems. Future work will broaden the analytic scope of modeling and verification methodologies, explore their applicability toward a broader range of safety-critical systems, and extend encoding tools to promote their widespread application in this design space.

List of References

[1] R. Shaw and M. T. Turvey, "Coalitions as models for ecosystems: A realist perspective on perceptual organization," *Perceptual Organization*, pp. 343–415, 1981.

[2] A. J. Wells, "Gibson's affordances and Turing's theory of computation," *Ecological Psychology*, vol. 14, no. 3, pp. 140–180, 2002.

[3] M. T. Turvey, "Affordances and prospective control: An outline of the ontology," *Ecological Psychology*, vol. 4, no. 3, pp. 173–187, 1992.

[4] T. A. Stoffregen, "Affordances as properties of the animal-environment system," *Ecological Psychology*, vol. 15, no. 2, pp. 115–134, 2003.

[5] H. Thiruvengada and L. Rothrock, "Using Petri nets for Gibson's affordances: First steps into perception-based task analysis," in *Proceedings of the Human Factors and Ergonomics Society 50th Annual Meeting.* Los Angeles, CA, USA: SAGE Publications, 2006, pp. 1132–1136.

[6] J. G. Greeno, "Gibson's affordances," *Psychological Review*, vol. 101, no. 2, pp. 336–342, 1994.

[7] A. Chemero, "An outline of a theory of affordances," *Ecological psychology*, vol. 15, no. 2, pp. 181–195, 2003.

[8] A. Lenarčič and M. Winter, "Affordances in situation theory," *Ecological Psychology*, vol. 25, no. 2, pp. 155–181, 2013.

[9] W. H. Warren, "Perceiving affordances: visual guidance of stair climbing." *Journal of Experimental Psychology: Human Perception and Performance*, vol. 10, no. 5, pp. 683–703, 1984.

[10] M. L. Bolton, R. I. Siminiceanu, and E. J. Bass, "A systematic approach to model checking human-automation interaction using task analytic models," *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 41, no. 5, pp. 961–976, 2011.

[11] EverMap LLC, "The AutoBookmark Plugin Tool," http://www.evermap.com/autobookmark.asp, 2016, accessed February 9, 2015.

[12] M. Egenhofer, "A model for detailed binary topological relationships," *Geomatica*, vol. 47, no. 3, pp. 261–273, 1993.

[13] T.-H. Trinh, P. Chevaillier, M. Barange, J. Soler, P. De Loor, and R. Querrec, "Integrating semantic directional relationships into virtual environments: A meta-modelling approach," in *Proceedings of the 17th Eurographics conference on Virtual Environments & Third Joint Virtual Reality.* Eurographics Association, 2011, pp. 67–74.

[14] A. Borrmann and E. Rank, "Specification and implementation of directional operators in a 3-D spatial query language for building information models," *Advanced Engineering Informatics*, vol. 23, no. 1, pp. 32–44, 2009.

[15] S. G. Chopski, O. M. Rangus, W. B. Moskowitz, and A. L. Throckmorton, "Experimental measurements of energy augmentation for mechanical circulatory assistance in a patient-specific Fontan model," *Artificial Organs*, vol. 38, no. 9, pp. 791–799, 2014.

[16] J. Bowen and V. Stavridou, "Safety-critical systems, formal methods and standards," *Software Engineering Journal*, vol. 8, no. 4, pp. 189–209, 1993.

[17] ISO, "ISO/IEC 62366-1:2015: Medical devices – Part 1: Application of usability engineering to medical devices," Geneva, 2015.

[18] ——, "ISO 92410-11: Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability," Geneva, 1998.

[19] C. Backinger and P. Kingsley, "Write it right: Recommendations for developing user instruction manuals for medical devices used in home health care (HHS Publication FDA 93-4258). Washington, DC: FDA," *Center for Devices and Radiological Health*, 1993.

[20] RTCA/EUROCAE, "DO-178C/ED-12C: Software considerations in airborne systems and equipment certification," Malakoff, France, 2012.

[21] U.S. FAA, "Electronic Flight Displays," U.S. Department of Transportation, Federal Aviation Administration, Washington, DC, Advisory Circular 25-11B, 2011.

[22] ——, "Airplane Flight Manual Change 1," U.S. Department of Transportation, Federal Aviation Administration, Washington, DC, USA, Document AC 25.1581-1, 2012.

[23] ISO, "IEC/TR 62366-2:2016: Medical devices – Part 2: Guidance on the application of usability engineering to medical devices," Geneva, 2016.

[24] U.S. FDA, "Applying Human Factors and Usability Engineering to Optimize Medical Device Design (Document number 1757)," *Center for Devices and Radiological Health*, 2016.

[25] ISO, "ISO/IEC CD 20071-23: Information technology – User interface component accessibility – Part 23: Guidance on the visual presentation of audio information (including captions and subtitles)," Geneva, 2007.

[26] ——, "ISO 5841-3:2013: Implants for surgery – Cardiac pacemakers – Part 3: Low-profile connectors (IS-1) for implantable pacemakers," Geneva, Switzerland, 2013.

[27] IEC, "IEC 61508: 2011: Functional safety of electrical/electronic/programmable electronic safety-related systems," Geneva, 2010.

[28] ISO, "ISO/IEC 60601-1-6:2010: Medical electrical equipment - Part 1-6: General requirements for basic safety and essential performance - Collateral standard: Usability," 2010.

[29] ——, "ISO/IEC 26514:2008: Systems and software engineering– Requirements for designers and developers of user documentation," Geneva, Switzerland, 2008.

[30] ——, "ISO 9355:2006: Ergonomic requirements for the design of displays and control actuators," Geneva, 2006.

[31] S. K. Card, A. Newell, and T. P. Moran, *The Psychology of Human-Computer Interaction.* Hillsdale, NJ, USA: Lawrence Erlbaum Associates Inc., 1983.

[32] J. Nielsen, "Usability inspection methods," in *Conference Companion on Human Factors in Computing Systems.* New York, NY, USA: ACM, 1994, pp. 413–414.

[33] D. A. Norman and S. W. Draper, *User Centered System Design; New Perspectives on Human-Computer Interaction.* Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1986.

[34] J. Nielsen, *Usability Inspection Methods.* New York, NY, USA: John Wiley & Sons, Inc., 1994, ch. Heuristic evaluation, pp. 25–62.

[35] ISO, "ISO 26262: 2011: Road vehicles – Functional safety," Geneva, Switzerland, 2011.

[36] D. A. Norman, "Some observations on mental models," *Mental Models*, vol. 7, no. 112, pp. 7–14, 1983.

[37] ——, "The way I see it: Signifiers, not affordances," *Interactions*, vol. 15, no. 6, pp. 18–19, 2008.

[38] ISO, "ISO 14971:2007: Medical devices – Application of risk management to medical devices," Geneva, 2007.

[39] E. Hollnagel, "The phenotype of erroneous actions," *International Journal of Man-Machine Studies*, vol. 39, no. 1, pp. 1–32, 1993.

[40] H. Hussmann, G. Meixner, and D. Zuehlke, *Model-Driven Development of Advanced User Interfaces*. Berlin, Germany: Springer Science & Business Media, 2011.

[41] J. M. Wing, "A specifier's introduction to formal methods," *Computer*, vol. 23, no. 9, pp. 8–22, 1990.

[42] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu, "Using formal verification to evaluate human-automation interaction: A review," *Systems, Man, and Cybernetics: Systems, IEEE Transactions on*, vol. 43, no. 3, pp. 488–503, 2013.

[43] J. C. Campos and M. D. Harrison, "Systematic analysis of control panel interfaces using formal tools," in *Proceedings of the 15th International Workshop on the Design, Verification and Specification of Interactive Systems*. New York, NY, USA: Springer, 2008, pp. 72–85.

[44] J. Gow, H. Thimbleby, and P. Cairns, "Automatic critiques of interface modes," in *International Workshop on Design, Specification, and Verification of Interactive Systems*. New York, NY, USA: Springer, 2005, pp. 201–212.

[45] G. D. Abowd, H. Wang, and A. F. Monk, "A formal technique for automated dialogue development," in *Proceedings of the 1st Conference on Designing Interactive Systems*. New York, NY, USA: ACM, 1995, pp. 219–226.

[46] E. J. Bass, K. M. Feigh, E. Gunter, and J. Rushby, "Formal modeling and analysis for interactive hybrid systems," in *Fourth International Workshop on Formal Methods for Interactive Systems: FMIS*, vol. 45. New York, NY, USA: Springer, 2011.

[47] A. Degani and M. Heymann, "Formal verification of human-automation interaction," *Human Factors: The Journal of the Human Factors and Ergonomics Society*, vol. 44, no. 1, pp. 28–43, 2002.

[48] M. S. Feary, "A toolset for supporting iterative human–automation interaction in design," NASA Ames Research Center, Tech. Rep. Technical Report 20100012861, 2010.

[49] J. S. Nandiganahalli, S. Lee, and I. Hwang, "Formal verification for mode confusion in the flight deck using intent-based abstraction," *Journal of Aerospace Information Systems*, vol. 13, no. 9, pp. 343–356, 2016.

[50] J. Rushby, J. Crow, and E. Palmer, "An automated method to detect potential mode confusions," in *Proceedings of the 18th Digital Avionics Systems Conference*. Piscataway, NJ, USA: IEEE, 1999, pp. 4.B.2–1–4.B.2–6.

[51] S. Pizziol, C. Tessier, M. Feary, and F. Dehais, "Action reversibility in human-machine systems," in *Proceedings of the 5th International Conference on Application and Theory of Automation in Command and Control Systems*, ser. ATACCS '15, 2015, pp. 21–31.

[52] K. Loer and M. Harrison, "Towards usable and relevant model checking techniques for the analysis of dependable interactive systems," in *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*. Piscataway, NJ, USA: IEEE, 2002, pp. 223–226.

[53] A. Hussey, I. MacColl, and D. Carrington, “Assessing usability from formal user-interface designs,” in *Proceedings of the 2001 Australian Software Engineering Conference*. Piscataway, NJ, USA: IEEE, 2001, pp. 40–47.

[54] M. J. Mahemoff and L. J. Johnston, “Principles for a usability-oriented pattern language,” in *Proceedings of the Australasian Conference on Computer-Human Interaction*. New York, NY, USA: Springer, 1998, pp. 132–139.

[55] R. Rukšėnas, P. Curzon, and A. Blandford, “Modelling and analysing cognitive causes of security breaches,” *Innovations in Systems and Software Engineering*, vol. 4, no. 2, pp. 143–160, 2008.

[56] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu, “Generating phenotypical erroneous human behavior to evaluate human–automation interaction using model checking,” *International Journal of Human-Computer Studies*, vol. 70, no. 11, pp. 888–906, 2012.

[57] M. L. Bolton, N. Jimenez, M. M. van Paassen, and M. Trujillo, “Automatically generating specification properties from task models for the formal verification of human–automation interaction,” *Human-Machine Systems, IEEE Transactions on*, vol. 44, no. 5, pp. 561–575, 2014.

[58] P. Palanque and F. Paternò, Eds., *Formal Methods in Human-Computer Interaction*. Secaucus, NJ, USA: Springer, 1997.

[59] J. C. Campos and M. D. Harrison, “Interaction engineering using the IVY tool,” in *Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. New York, NY, USA: ACM, 2009, pp. 35–44.

[60] Y. Aït-Ameur and M. Baron, “Formal and experimental validation approaches in HCI systems design based on a shared event B model,” *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 6, pp. 547–563, 2006.

[61] P. Curzon, R. Rukšėnas, and A. Blandford, “An approach to formal verification of human–computer interaction,” *Formal Aspects of Computing*, vol. 19, no. 4, pp. 513–550, 2007.

[62] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[63] N. Guerreiro, S. Mendes, V. Pinheiro, and J. C. Campos, “AniMAL-a user interface prototyper and animator for MAL interactor models,” in *Interacção 2008–Actas da 3a. Conferência Nacional Interacção Pessoa-Máquina*. Grupo Português de Computação Gráfica, 2008, pp. 93–102.

[64] S. Combéfis, D. Giannakopoulou, C. Pecheur, and M. Feary, “A formal framework for design and analysis of human-machine interaction,” in *2011 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. Piscataway, NJ, USA: IEEE, 2011, pp. 1801–1808.

[65] S. Combéfis, D. Giannakopoulou, and C. Pecheur, “Automatic detection of potential automation surprises for adept models,” *IEEE Transactions on Human-Machine Systems*, vol. 46, no. 2, pp. 267–278, 2016.

[66] F. Paternò, C. Mancini, and S. Meniconi, “ConcurTaskTrees: A diagrammatic notation for specifying task models,” in *Human-Computer Interaction INTERACT97*. New York, NY, USA: Springer, 1997, pp. 362–369.

[67] Y. Aït-Ameur, B. Breholee, P. Girard, L. Guittet, and F. Jambon, “Formal verification and validation of interactive systems specifications,” in *Proceedings of the 7th Working Conference on Human Error, Safety and Systems Development*, C. W. Johnson and P. A. Palanque, Eds. New York, NY, USA: Springer, 2004, pp. 61–76.

[68] L. De Moura, S. Owre, and N. Shankar, "The SAL language manual," Computer Science Laboratory, SRI International, Menlo Park, Tech. Rep. CSL-01-01, 2003.

[69] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[70] M. Heymann and A. Degani, "Automated driving aids: modeling, analysis, and interface design considerations," in *Proceedings of the 5th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*. New York, NY, USA: ACM, 2013, pp. 142–149.

[71] L. Quin, "Extensible Markup Language 1.0," 1997, http://www.w3.org/{XML}/Core/.

[72] J. Reason, *Human Error*. New York, NY, USA: Cambridge University Press, 1990.

[73] M. L. Bolton and E. J. Bass, "Using task analytic models to visualize model checker counterexamples," in *Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on*. Piscataway, NJ, USA: IEEE, 2010, pp. 2069–2074.

[74] ——, "Using model checking to explore checklist-guided pilot behavior," *The International Journal of Aviation Psychology*, vol. 22, no. 4, pp. 343–366, 2012.

[75] J. C. Campos, M. Sousa, M. C. B. Alves, and M. D. Harrison, "Formal verification of a space system's user interface with the IVY workbench," *IEEE Transactions on Human-Machine Systems*, vol. 46, no. 2, pp. 303–316, 2016.

[76] G. Doherty and M. Massink, "Continuous interaction and human control," in *Proceedings of the XVIII European Annual Conference on Human Decision Making and Manual Control*. New York, NY, USA: Springer, 1999, pp. 80–96.

[77] G. Doherty, M. Massink, and G. Faconti, "Using hybrid automata to support human factors analysis in a critical system," *Formal Methods in System Design*, vol. 19, no. 2, pp. 143–164, 2001.

[78] P. Bagade, A. Banerjee, and S. K. Gupta, "Safety assurance of medical cyber-physical systems using hybrid automata: A case study on analgesic infusion pump," in *Medical Cyber Physical Systems Workshop*. New York, NY, USA: Springer, 2013.

[79] A. Schwarze, M. Buntins, J. Schicke-Uffmann, U. Goltz, and F. Eggert, "Modelling driving behaviour using hybrid automata," *IET Intelligent Transport Systems*, vol. 7, no. 2, pp. 251–256, 2013.

[80] S. K. Card, T. P. Moran, and A. Newell, "The keystroke-level model for user performance time with interactive systems," *Communications of the ACM*, vol. 23, no. 7, pp. 396–410, 1980.

[81] A. Degani and E. L. Wiener, "Philosophy, policies, procedures and practices- the four'p's of flight deck operations," *Aviation Psychology in Practice (A 96-10209 01-53), Aldershot, United Kingdom, Avebury Technical, 1994,*, pp. 44–67, 1994.

[82] J. J. Gibson, *The Ecological Approach to Visual Perception*. Boston, MA, USA: Houghton Mifflin, 1979.

[83] U.S. FDA, "Manufacturer and User Facility Device Experience Database – (MAUDE)," http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/PostmarketRequirements/ReportingAdverseEvents/ucm127891.htm, Washington D.C., USA, 2016, accessed: July 25, 2016.

[84] T. D. Griffiths, C. Büchel, R. S. Frackowiak, and R. D. Patterson, "Analysis of temporal structure in sound by the human brain," *Nature Neuroscience*, vol. 1, no. 5, pp. 422–427, 1998.

[85] E. Pöppel, "A hierarchical model of temporal perception," *Trends in Cognitive Sciences*, vol. 1, no. 2, pp. 56–61, 1997.

[86] P. Cairns and H. Thimbleby, "Affordance and symmetry in user interfaces," *The Computer Journal*, vol. 51, no. 6, pp. 650–661, 2008.

[87] D. H. Kafagy, T. W. Dwyer, K. L. McKenna, J. P. Mulles, S. G. Chopski, W. B. Moskowitz, and A. L. Throckmorton, "Design of axial blood pumps for patients with dysfunctional Fontan physiology: Computational studies and performance testing," *Artificial Organs*, vol. 39, no. 1, pp. 34–42, 2015.

[88] A. Newell, *Unified Theories of Cognition*. Cambridge, MA, USA: Harvard University Press, 1994.

[89] L. W. Barsalou, "Perceptions of perceptual symbols," *Behavioral and Brain Sciences*, vol. 22, no. 04, pp. 637–660, 1999.

[90] A. Blandford, R. Butterworth, and J. Good, "Users as rational interacting agents: Formalising assumptions about cognition and interaction," in *Proceedings of the 4th International Eurographics Workshop, on Design, Specification and Verification of Interactive Systems*, vol. 97. New York, NY, USA: Springer, 1997, pp. 45–60.

[91] B. Kirwan and L. K. Ainsworth, *A Guide to Task Analysis*. London, England, U.K.: Taylor and Francis, 1992.

[92] J. Annett, "Hierarchical task analysis," *Handbook of Cognitive Task Design*, vol. 2, pp. 17–35, 2003.

[93] T. De Jong and M. G. Ferguson-Hessler, "Types and qualities of knowledge," *Educational Psychologist*, vol. 31, no. 2, pp. 105–113, 1996.

[94] RTCA/EUROCAE, "DO-333/ED-218: Formal methods supplement," Malakoff, France, 2012.

[95] U.S. DOD, "MIL-STD-882E: Standard practice for system safety," Washington D.C., USA, 2012.

[96] P. Palanque and C. Martinie, "Designing and assessing interactive systems using task models," in *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, 2016, pp. 976–979.

[97] C. Martinie, P. Palanque, R. Fahssi, J.-P. Blanquart, C. Fayollas, and C. Seguin, "Task model-based systematic analysis of both system failures and human errors," *IEEE Transactions on Human-Machine Systems*, vol. 46, no. 2, pp. 243–254, 2016.

[98] A. Degani, M. Heymann, and M. Shafto, "Modeling and formal analysis of human-machine interaction," *The Oxford Handbook of Cognitive Engineering*, pp. 433–448, 2013.

[99] P. Palanque, R. Bastide, and V. Sengès, "Validating interactive system design through the verification of formal task models," *Engineering for HCI*, pp. 189–210, 2016.

[100] C. Muñoz, V. Carreño, and G. Dowek, "Formal analysis of the operational concept for the Small Aircraft Transportation System," in *Rigorous Development of Complex Fault-Tolerant Systems*. New York, NY, USA: Springer, 2006, pp. 306–325.

[101] P. Masci, R. Rukšėnas, P. Oladimeji, A. Cauchi, A. Gimblett, Y. Li, P. Curzon, and H. Thimbleby, "The benefits of formalising design guidelines: A case study on the predictability of drug infusion pumps," *Innovations in Systems and Software Engineering*, pp. 1–21, 2013.

[102] J.-K. Joo and N.-H. Kim, "Modeling and simulation of emergent evacuation using affordance-based fsa models," *Journal of Korean Institute of Industrial Engineers*, vol. 37, no. 2, pp. 96–104, 2011.

[103] J. Crow, D. Javaux, and J. Rushby, "Models and mechanized methods that integrate human factors into automation design," in *Proceedings of the 2000 International Conference on Human-computer Interaction in Aeronautics*. New York, NY, USA: ACM, 2000, pp. 163–168.

[104] D. L. Parnas, "On the use of transition diagrams in the design of a user interface for an interactive computer system," in *Proceedings of the 24th National ACM Conference*. New York, NY, USA: ACM, 1969, pp. 379–385.

[105] N. G. Leveson, L. D. Pinnel, S. D. Sandys, S. K., and J. D. Reese, "Analyzing software specifications for mode confusion potential," in *Proceedings of the Workshop on Human Error and System Development*. Glasgow: University of Glasgow, 1997, pp. CD–ROM.

[106] K. L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*. Pittsburgh, PA, USA: Carnegie Mellon University, 1993.

[107] C. Heitmeyer, "On the need for practical formal methods," in *Proceedings of the 5th International Symposium on Formal Techniques in Real-Time Fault-Tolerant Systems*. New York, NY, USA: Springer, 1998, pp. 18–26.

[108] P. Masci, Y. Zhang, P. Jones, P. Curzon, and H. Thimbleby, "Formal verification of medical device user interfaces using PVS," in *Fundamental Approaches to Software Engineering*. New York, NY, USA: Springer, 2014, pp. 200–214.

[109] J. C. Campos and M. D. Harrison, "Formally verifying interactive systems: A review," in *Design, Specification and Verification of Interactive Systems*, 1997, pp. 109–124.

[110] R. Butterworth, A. Blandford, and D. Duke, "Using formal models to explore display-based usability issues," *Journal of Visual Languages and Computing*, vol. 10, no. 5, pp. 455–479, 1999.

[111] R. C. Boyatt and J. E. Sinclair, "A "lightweight formal methods" perspective on investigating aspects of interactive systems," in *Pre-proceedings of the 2nd International Workshop on Formal Methods for Interactive Systems*. New York, NY, USA: Springer, 2007, pp. 35–50.

[112] P. Masci, P. Curzon, A. Blandford, and D. Furniss, "Modelling distributed cognition systems in pvs," *Electronic Communications of the EASST*, vol. 45, 2011.

[113] R. D. Hill, "Event-response systems: a technique for specifying multi-threaded dialogues," *ACM SIGCHI Bulletin*, vol. 18, no. 4, pp. 241–248, 1987.

[114] T. A. Henzinger, "The theory of hybrid automata," in *Verification of Digital and Hybrid Systems*. New York, NY, USA: Springer, 2000, pp. 265–292.

[115] L. Rothrock, R. Wysk, N. Kim, D. Shin, Y.-J. Son, and J. Joo, "A modelling formalism for human-machine cooperative systems," *International Journal of Production Research*, vol. 49, no. 14, pp. 4263–4273, 2011.

[116] H. Thimbleby, A. Gimblett, and A. Cauchi, "Buffer automata: A UI architecture prioritising HCI concerns for interactive devices," in *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. New York, NY, USA: ACM, 2011, pp. 73–78.

[117] R. Alur, T. Henzinger, G. Lafferriere, and G. Pappas, "Discrete abstractions of hybrid systems," *Proceedings of the IEEE*, vol. 88, no. 7, pp. 971–984, 2000.

[118] S. Sankaranarayanan and A. Tiwari, "Relational abstractions for continuous and hybrid systems," in *International Conference on Computer Aided Verification*. New York, NY, USA: Springer, 2011, pp. 686–702.

[119] A. Tiwari and G. Khanna, "Series of abstractions for hybrid automata," in *International Workshop on Hybrid Systems: Computation and Control*. New York, NY, USA: Springer, 2002, pp. 465–478.

[120] J. Lygeros, K. H. Johansson, S. N. Simic, J. Zhang, and S. S. Sastry, "Dynamical properties of hybrid automata," *IEEE Transactions on Automatic Control*, vol. 48, no. 1, pp. 2–17, 2003.

[121] A. Banerjee, Y. Zhang, P. Jones, and S. Gupta, "Using formal methods to improve home-use medical device safety," *Biomedical Instrumentation & Technology*, vol. 47, no. 1, p. 43, 2013.

[122] S. J. Payne and T. R. Green, "Task-action grammars: A model of the mental representation of task languages," *Human-Computer Interaction*, vol. 2, no. 2, pp. 93–133, 1986.

[123] H. R. Hartson, A. C. Siochi, and D. Hix, "The UAN: A user-oriented representation for direct manipulation interface designs," *ACM Transactions on Information Systems*, vol. 8, no. 3, pp. 181–203, 1990.

[124] B. E. John and D. E. Kieras, "Using GOMS for user interface design and evaluation: Which technique?" *ACM Transactions Computer-Human Interaction*, vol. 3, no. 4, pp. 287–319, 1996.

[125] D. Harel and A. Pnueli, "On the development of reactive systems," in *Logics and Models of Concurrent Systems*. New York, NY, USA: Springer, 1985, pp. 477–498.

[126] C. Martinie, P. Palanque, E. Barboni, M. Winckler, M. Ragosta, A. Pasquini, and P. Lanzi, "Formal tasks and systems models as a tool for specifying and assessing automation designs," in *Proceedings of the 1st International Conference on Application and Theory of Automation in Command and Control Systems*. Barcelona, Spain: IRIT Press, 2011, pp. 50–59.

[127] A. Dix, M. Ghazali, S. Gill, J. Hare, and D. Ramduny-Ellis, "Physigrams: Modelling devices for natural interaction," *Formal Aspects of Computing*, pp. 1–29, 2008.

[128] W. Damm, S. Disch, H. Hungar, S. Jacobs, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz, "Exact state set representations in the verification of linear hybrid systems with large discrete state space," in *International Symposium on Automated Technology for Verification and Analysis*. New York, NY, USA: Springer, 2007, pp. 425–440.

[129] J. R. Dormand and P. J. Prince, "A family of embedded Runge-Kutta formulae," *Journal of Computational and Applied Mathematics*, vol. 6, no. 1, pp. 19–26, 1980.

[130] P. A. Fuhrmann, "On weak and strong reachability and controllability of infinite-dimensional linear systems," *Journal of Optimization Theory and Applications*, vol. 9, no. 2, pp. 77–89, 1972.

[131] J. R. Anderson, M. Matessa, and C. Lebiere, "ACT-R: A theory of higher level cognition and its relation to visual attention," *Human-Computer Interaction*, vol. 12, no. 4, pp. 439–462, 1997.

[132] A. Wise, "Little-JIL 1.0 language report," Technical Report 98-24, University of Massachusetts at Amherst, Tech. Rep., 1998.

[133] P. Masci, Y. Zhang, P. Jones, P. Oladimeji, E. DUrso, C. Bernardeschi, P. Curzon, and H. Thimbleby, "Combining PVSio with stateflow," in *NASA Formal Methods Symposium*. New York, NY, USA: Springer, 2014, pp. 209–214.

[134] C. Martinie, P. Palanque, M. Ragosta, and R. Fahssi, "Extending procedural task models by systematic explicit integration of objects, knowledge and information," in *Proceedings of the 31st European Conference on Cognitive Ergonomics*. New York, NY, USA: ACM, 2013, p. 23.

[135] J. Barnett, R. Akolkar, R. Auburn, M. Bodell, D. C. Burnett, J. Carter, S. McGlashan, T. Lager, M. Helbing, R. Hosn *et al.*, “State chart XML (scXML): State machine notation for control abstraction,” W3C, Tech. Rep., 2007.

[136] J. Gow and H. Thimbleby, “MAUI: An interface design tool based on matrix algebra,” in *Computer-Aided Design of User Interfaces IV*. New York, NY, USA: Springer, 2005, pp. 81–94.

[137] D. Navarre, P. Palanque, J.-F. Ladry, and E. Barboni, “Icos: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability,” *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 16, no. 4, p. 18, 2009.

[138] P. A. Akiki, A. K. Bandara, and Y. Yu, “Cedar studio: an IDE supporting adaptive model-driven user interfaces for enterprise applications,” in *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. New York, NY, USA: ACM, 2013, pp. 139–144.

[139] N. Almeida, S. Silva, and A. Teixeira, “Multimodal multi-device application supported by an scXML state chart machine,” in *Proceedings of the First EICS Workshop on Engineering Interactive Computer Systems with SCXML*. New York, NY, USA: Springer, 2014, pp. 12–17.

[140] P. Masci, P. Oladimeji, Y. Zhang, P. Jones, P. Curzon, and H. Thimbleby, “PVSio-web 2.0: Joining PVS to HCI,” in *Computer Aided Verification*. New York, NY, USA: Springer, 2015, pp. 470–478.

[141] F. Paternò and C. Santoro, “Integrating model checking and HCI tools to help designers verify user interface properties,” in *Proceedings of the 7th International Workshop on Design, Specification, and Verification of Interactive Systems*. Berlin, Germany: Springer-Berlin, 2001, pp. 135–150.

[142] E. Barboni, J.-F. Ladry, D. Navarre, P. Palanque, and M. Winckler, “Beyond modelling: an integrated environment supporting co-execution of tasks and systems models,” in *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. New York, NY, USA: ACM, 2010, pp. 165–174.

[143] A. J. Abbate, E. J. Bass, and A. L. Throckmorton, “A formal task analytic approach to medical device alarm troubleshooting instructions,” *IEEE Transactions on Human-Machine Systems*, vol. 41, no. 1, pp. 53–65, 2016.

[144] A. Tiwari, “HybridSAL relational abstracter,” in *Proceedings of the 24th International Conference on Computer Aided Verification*. New York, NY, USA: Springer, 2012, pp. 725–731.

[145] J. Rushby, “Using model checking to help discover mode confusions and other automation surprises,” *Reliability Engineering and System Safety*, vol. 75, no. 2, pp. 167–177, 2002.

[146] H. Thimbleby, “User interface design with matrix algebra,” *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 11, no. 2, pp. 181–236, 2004.

[147] M. Duflot, M. Kwiatkowska, G. Norman, D. Parker, S. Peyronnet, C. Picaronny, and J. Sproston, “Practical applications of probabilistic model checking to communication protocols,” *Formal Methods for Industrial Critical Systems: A Survey of Applications*, pp. 133–150, 2012.

[148] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao, “Efficient generation of counterexamples and witnesses in symbolic model checking,” in *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*. New York, NY, USA: ACM, 1995, pp. 427–432.

[149] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, “Bounded model checking,” *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[150] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. 100, no. 6, pp. 509–516, 1978.

[151] J. R. Burch, E. M. Clarke, D. L. Dill, J. Hwang, and K. L. McMillan, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–171, 1992.

[152] S. Kripke, "Semantical considerations of the modal logic," *Studia Philosophica*, vol. 1, no. 9, pp. 67–96, 1972.

[153] J. M. Spivey and J. Abrial, *The Z Notation*. Hemel Hempstead, U.K.: Prentice Hall, 1992.

[154] N. Shankar, "Symbolic analysis of transition systems," in *Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2000, pp. 287–302.

[155] P. Gastin and D. Oddoux, "Fast LTL to Büchi automata translation," in *International Conference on Computer Aided Verification*. New York, NY, USA: Springer, 2001, pp. 53–65.

[156] M. L. Bolton, X. Zheng, K. Molinaro, A. Houser, and M. Li, "Improving the scalability of formal human–automation interaction verification analyses that use task-analytic models," *Innovations in Systems and Software Engineering*, pp. 1–17, 2016.

[157] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories." *Handbook of Satisfiability*, vol. 185, pp. 825–885, 2009.

[158] J. Clark and M. Murata, "Relax NG specification," Committee Specification, Organization for the Advancement of Structured Information Standards, 2001. [Online]. Available: http://relaxng.org/spec-20011203.html

[159] C. Sperberg-McQueen and H. Thompson, "The W3C XML Schema 1.0," 2001, http://www.w3.org/{XML}/Schema.

[160] A. J. Abbate and E. J. Bass, "Using computational tree logic methods to analyze reachability in user documentation," in *Proceedings of the 2015 Annual Meeting of the Human Factors and Ergonomics Society*. Los Angeles, CA, USA: SAGE Publications, 2015, pp. 1481–1485.

[161] U.S. FAA, "Aviation Safety Reporting System," U.S. Department of Transportation, Federal Aviation Administration, Washington, DC, Advisory Circular 00-46E, 2011.

[162] S. Bly and C. C. Marshall, "Turning the page on navigation," in *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries*. New York, NY, USA: ACM, 2005, pp. 225–234.

[163] J. Alexander and A. Cockburn, "An empirical characterisation of electronic document navigation," in *Proceedings of Graphics Interface 2008*, ser. GI '08. New York, NY, USA: ACM, 2008, pp. 123–130.

[164] V. Liesaputra and I. H. Witten, "Seeking information in realistic books: a user study," in *Proceedings of the 8th ACM/IEEE-CS Joint Conference on Digital libraries*. New York, NY, USA: ACM, 2008, pp. 29–38.

[165] N. Shankar and M. Sorea, "Counterexample-driven model checking," Computer Science Laboratory, SRI International, Menlo Park, Tech. Rep. SRI-CSL-03-04, 2003.

[166] M. L. Bolton and E. J. Bass, "Formally verifying human-automation interaction as part of a system model: Limitations and tradeoffs," *Innovations in Systems and Software Engineering: A NASA Journal*, vol. 6, no. 3, pp. 219–231, 2010.

[167] ——, "Building a formal model of a human-interactive system: Insights into the integration of formal methods and human factors engineering," in *Proceedings of the 1st NASA Formal Methods Symposium*, 2009, pp. 6–15.

[168] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[169] J. André, "Can structured formatters prevent train crashes?" *Electronic Publishing*, vol. 2, no. 3, pp. 169–173, 1988.

[170] M. Addison and H. Thimbleby, "Manuals as structured programs," in *Proceedings of the Conference on People and Computers IX*. Cambridge, U.K.: Cambridge University Press, 1994, pp. 67–79.

[171] H. Thimbleby and M. Addison, "Hyperdoc: an interactive systems tool," in *Proceedings of the HCI'95 Conference on People and Computers X*. Cambridge, U.K.: Cambridge University Press, 1996, pp. 95–106.

[172] H. Thimbleby and P. Ladkin, "From logic to manuals again," *IEE Proceedings–Software Engineering*, vol. 144, no. 3, pp. 185–192, 1997.

[173] H. Thimbleby and P. B. Ladkin, "A proper explanation when you need one," in *Proceedings of the HCI'95 Conference on People and Computers X*, ser. HCI '95. New York, NY, USA: ACM, 1995, pp. 107–118.

[174] H. Thimbleby, "User-centered methods are insufficient for safety critical systems," in *Proceedings of the 3rd Human-Computer Interaction and Usability Engineering of the Austrian Computer Society Conference on HCI and Usability for Medicine and Health Care*. New York, NY, USA: Springer, 2007, pp. 1–20.

[175] H. Thimbleby and M. Addison, "Intelligent adaptive assistance and its automatic generation," *Interacting with Computers*, vol. 8, no. 1, pp. 51–68, 1996.

[176] H. Thimbleby, "Interaction walkthrough: evaluation of safety critical interactive systems," in *Interactive Systems. Design, Specification, and Verification*. New York, NY, USA: Springer, 2007, pp. 52–66.

[177] G. V. Bochmann and C. A. Sunshine, "A survey of formal methods," in *Computer Network Architectures and Protocols*. New York, NY, USA: Springer, 1982, pp. 561–578.

[178] O. Kupferman and M. Y. Vardi, "Model checking of safety properties," *Formal Methods in System Design*, vol. 19, no. 3, pp. 291–314, 2001.

[179] J. Rushby, "The versatile synchronous observer," in *Specification, Algebra, and Software*. New York, NY, USA: Springer, 2014, pp. 110–128.

[180] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[181] D. Stamatis, *Failure Mode and Effect Analysis: FMEA from Theory to Execution*. Milwaukee, WI, USA: ASQ Quality Press, 2003.

[182] D. A. Norman, "Affordance, conventions, and design," *Interactions*, vol. 6, no. 3, pp. 38–43, 1999.

[183] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Journal of Math*, vol. 58, no. 345-363, p. 5, 1936.

[184] N. Kim, L. Rothrock, J. Joo, and R. Wysk, "An affordance-based formalism for modeling human-involvement in complex systems for prospective control," in *Simulation Conference (WSC), Proceedings of the 2010 Winter*. Piscataway, NJ, USA: IEEE, 2010, pp. 811–823.

[185] K. Jensen, "Coloured Petri nets and the invariant-method," *Theoretical Computer Science*, vol. 14, no. 3, pp. 317–336, 1981.

[186] B. P. Zeigler, *Multifacetted Modelling and Discrete Event Simulation*. San Diego, CA, USA: Academic Press Professional, Inc., 1984.

[187] J. Barwise and J. Perry, "Situations and attitudes," *The Journal of Philosophy*, pp. 668–691, 1981.

[188] ISO, "ISO 10303-11:2004: Industrial automation systems and integration – Product data representation and exchange – Part 11: Description methods: The EXPRESS language reference manual," Geneva, 2004.

[189] ——, "ISO 8807:1989: Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour," Geneva, 1989.

[190] L. Talmy, *How Language Structures Space*. New York, NY, USA: Pelnum Press, 1983.

[191] K. Ingram and W. Phillips, "Geographic information processing using a SQL-based query language," in *Proceedings of Auto-Carto*, vol. 8, 1987, pp. 326–335.

[192] J. Herring, R. Larsen, and J. Shivakumar, "Extensions to the SQL language to support spatial analysis in a topological data base," in *Proceedings of GIS/LIS*, vol. 88. New York, NY, USA: Springer, 1988, pp. 741–750.

[193] B. C. Ooi, R. Sacks-Davis, and K. J. McDonell, "Extending a DBMS for geographic applications." in *ICDE*, 1989, pp. 590–597.

[194] N. Roussopoulos, C. Faloutsos, and T. Sellis, "An efficient pictorial database system for PSQL," *Software Engineering, IEEE Transactions on*, vol. 14, no. 5, pp. 639–650, 1988.

[195] M. J. Egenhofer, "Spatial SQL: A query and presentation language," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 6, no. 1, pp. 86–95, 1994.

[196] A. U. Frank, "Qualitative spatial reasoning about distances and directions in geographic space," *Journal of Visual Languages & Computing*, vol. 3, no. 4, pp. 343–371, 1992.

[197] M. J. Egenhofer and R. D. Franzosa, "Point-set topological spatial relations," *International Journal of Geographical Information System*, vol. 5, no. 2, pp. 161–174, 1991.

[198] M. J. Egenhofer and J. Herring, "A mathematical framework for the definition of topological relationships," in *Fourth International Symposium on Spatial Data Handling*. IGU Commission of GIS, 1990, pp. 803–813.

[199] D. J. Peuquet and Z. Ci-Xiang, "An algorithm to determine the directional relationship between arbitrarily-shaped polygons in the plane," *Pattern Recognition*, vol. 20, no. 1, pp. 65–74, 1987.

[200] A. Borrmann, C. Van Treeck, and E. Rank, "Towards a 3-D spatial query language for building information models," in *Proceedings of the Joint International Conference on Computing and Decision Making in Civil and Building Engineering (ICCCBE-XI)*, vol. 2. New York, NY, USA: Springer, 2006.

[201] J. Bird and C. Ross, *Mechanical Engineering Principles*. New York, NY, USA: Routledge, 2012.

[202] H. Thiruvengada and L. Rothrock, "HCI implications of a Colored Petri Net model for Gibson's affordance," in *IIE Annual Conference Proceedings*. Institute of Industrial Engineers, 2006, pp. 1–6.

[203] U.S. FDA, "Medical Device Reporting (MDR)," http://www.fda.gov/MedicalDevices/Safety/ReportaProblem/default.htm, Washington D.C., USA, 2015, accessed: September 25, 2016.

[204] B. Scientific, "Physician's Technical Manual: VALITUDE, VALITUDE X4, INTUA, INVIVE," https://www.bostonscientific.com/content/dam/Manuals/us/current-rev-en/359252-001_Ingenio1-2_CRT-P_PTM_en-USA_S.pdf, St. Paul, MN, U.S.A., 2014, accessed May 23, 2016.

[205] A. J. Abbate and E. J. Bass, "A formal language for specifying visual interface signifiers," in *Proceedings of the 2016 Annual Meeting of the Human Factors and Ergonomics Society*. Los Angeles, CA, USA: SAGE Publications, 2016, pp. 1148–1153.

[206] C. S. De Souza, *The Semiotic Engineering of Human-Computer Interaction*. Cambridge, MA, USA: MIT Press, 2005.

[207] U.S. FDA, "MAUDE report 3003768277-2015-00048," http://www.fda.gov/MedicalDevices/Safety/ReportaProblem/default.htm, Washington D.C., USA, 2015, accessed June 23, 2016.

[208] Philips, "Interventional X-ray," http://www.usa.philips.com/healthcare/solutions/interventional-xray, 2011, accessed: October 21, 2016.

[209] C. Perrow, *Normal Accidents: Living with High-Risk Technologies*. Princeton, NJ, USA: Princeton University Press, 1999.

[210] C.-Y. Chan, "Defining safety performance measures of driver-assistance systems for intersection left-turn conflicts," in *2006 IEEE Intelligent Vehicles Symposium*. Piscataway, NJ, USA: IEEE, 2006, pp. 25–30.

[211] E. C. Haas and J. B. Van Erp, "Multimodal warnings to enhance risk communication and safety," *Safety Science*, vol. 61, pp. 29–35, 2014.

[212] C. S. De Souza, C. F. Leitão, R. O. Prates, and E. J. da Silva, "The semiotic inspection method," in *Proceedings of the VII Brazilian Symposium on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2006, pp. 148–157.

[213] J. Goguen, *An Introduction to Algebraic Semiotics, with Application to User Interface Design*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 1999, pp. 242–291.

[214] S. Sankaranarayanan, H. Homaei, and C. Lewis, "Model-based dependability analysis of programmable drug infusion pumps," in *International Conference on Formal Modeling and Analysis of Timed Systems*. New York, NY, USA: Springer, 2011, pp. 317–334.

[215] G. Gelman, K. M. Feigh, and J. Rushby, "Example of a complementary use of model checking and human performance simulation," *IEEE Transactions on Human-Machine Systems*, vol. 44, no. 5, pp. 576–590, 2014.

[216] A. V. Kaplan, D. S. Baim, J. J. Smith, D. A. Feigal, M. Simons, D. Jefferys, T. J. Fogarty, R. E. Kuntz, and M. B. Leon, "Medical device development from prototype to regulatory approval," *Circulation*, vol. 109, no. 25, pp. 3068–3072, 2004.

[217] S. G. Chopski, O. M. Rangus, E. A. Downs, W. B. Moskowitz, and A. L. Throckmorton, "Three-dimensional laser flow measurements of a patient-specific Fontan physiology with mechanical circulatory assistance," *Artificial Organs*, vol. 39, no. 6, pp. E67–E78, 2015.

[218] R. B. M. Hosein, A. J. B. Clarke, S. P. McGuirk, M. Griselli, O. Stumper, J. V. De Giovanni, D. J. Barron, and W. J. Brawn, "Factors influencing early and late outcome following the Fontan procedure in the current era. The 'Two Commandments'?" *European Journal of Cardio-thoracic Surgery*, vol. 31, no. 3, pp. 344–352, 2007.

[219] A. L. Throckmorton and S. G. Chopski, "Pediatric circulatory support: current strategies and future directions. biventricular and univentricular mechanical assistance," *ASAIO Journal*, vol. 54, no. 5, pp. 491–497, 2008.

[220] ANSYS Inc., "Computational Fluid Dynamics: CFX," http://www.ansys.com/products/Fluids/ANSYS-CFX, 2016, accessed November 16, 2016.

[221] N. H. Anderson, *A Functional Theory of Cognition.* New York, NY, USA: Psychology Press, 2014.

[222] U.S. FDA, "Expedited Access for Premarket Approval and De Novo Medical Devices Intended for Unmet Medical Needfor Life Threatening or Irreversibly Debilitating Diseases or Conditions," *Center for Devices and Radiological Health*, 2015.

[223] P. Legris, J. Ingham, and P. Collerette, "Why do people use information technology? a critical review of the technology acceptance model," *Information & Management*, vol. 40, no. 3, pp. 191–204, 2003.

[224] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An open source tool for symbolic model checking," in *International Conference on Computer Aided Verification.* New York, NY, USA: Springer, 2002, pp. 359–364.

[225] L. De Moura, H. Rueß, and M. Sorea, "Bounded model checking and induction: From refutation to verification," in *International Conference on Computer Aided Verification.* New York, NY, USA: Springer, 2003, pp. 14–26.

[226] M. B. Dwyer, Robby, O. Tkachuk, and W. Visser, "Analyzing interaction orderings with model checking," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering.* Piscataway, NJ, USA: IEEE, 2004, pp. 154–163.

[227] A. Carlisle and F. Siddi, "Blender Reference Manual," 2016, http://docs.blender.org/manual/en/dev//.

[228] P. S. Foundation, "Python Programming Language," 2008, http://www.python.org.

[229] M. Li, K. Molinaro, and M. L. Bolton, "Learning formal human-machine interface designs from task analytic models," in *Proceedings of the 2015 Human Factors and Ergonomics Society Annual Meeting*, vol. 59, no. 1. Los Angeles, CA, USA: SAGE Publications, 2015, pp. 652–656.

[230] A. G. Cass, A. Lerner, E. K. McCall, L. J. Osterweil, S. M. Sutton, and A. Wise, "Little-JIL/Juliette: a process definition language and interpreter," in *Software Engineering, 2000. Proceedings of the 2000 International Conference on.* Piscataway, NJ, USA: IEEE, 2000, pp. 754–757.

[231] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich, "Flow analysis for verifying properties of concurrent software systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 13, no. 4, pp. 359–430, 2004.

[232] H. M. Conboy, G. S. Avrunin, and L. A. Clarke, "Modal abstraction view of requirements for medical devices used in healthcare processes," in *Software Engineering in Health Care (SEHC), 2013 5th International Workshop on.* Piscataway, NJ, USA: IEEE, 2013, pp. 24–27.

[233] A. Gawanmeh, H. Al-Hamadi, M. Al-Qutayri, S.-K. Chin, and K. Saleem, "Reliability analysis of healthcare information systems: State of the art and future directions," in *2015 17th International Conference on E-health Networking, Application & Services (HealthCom)*. Piscataway, NJ, USA: IEEE, 2015, pp. 68–74.

[234] J. L. Silva, J. C. Campos, and A. C. Paiva, "Model-based user interface testing with Spec Explorer and ConcurTaskTrees," *Electronic Notes in Theoretical Computer Science*, vol. 208, pp. 77–93, 2008.

[235] J. L. Silva, O. R. Ribeiro, J. M. Fernandes, J. C. Campos, and M. D. Harrison, "The APEX framework: prototyping of ubiquitous environments based on Petri nets," in *International Conference on Human-Centred Software Engineering*. New York, NY, USA: Springer, 2010, pp. 6–21.

[236] E. Serral, J. De Smedt, and J. Vanthienen, "Correctness checking of pervasive behaviour by mapping task models to petri nets," *International SERIES on Information Systems and Management in Creative eMedia*, vol. 17, no. 1, pp. 11–18, 2015.

[237] A. Chebieb and Y. A. Ameur, "Formal verification of plastic user interfaces exploiting domain ontologies," in *Theoretical Aspects of Software Engineering (TASE), 2015 International Symposium on*. Piscataway, NJ, USA: IEEE, 2015, pp. 79–86.

[238] C. Martinie, P. Palanque, and M. Winckler, "Structuring and composition mechanisms to address scalability issues in task models," in *IFIP Conference on Human-Computer Interaction*, 2011, pp. 589–609.

[239] C. G. Williams, "Using concept maps to assess conceptual knowledge of function," *Journal for Research in Mathematics Education*, pp. 414–421, 1998.

Appendix A: List of Symbols

Name	Description	Symbol	Page(s)
Always	Specifies that one or more logical axioms must always be valued *true* for a temporal logic specification	G	29, 30, 87, 88, 153, 154, 168, 196–199, 225, 244, 260–273
Delta	Uppercase Greek letter delta specifying a change in variable values for a Z schema identified on the right-hand side	Δ	44, 63, 193
Empty set	Represents a set containing no elements	$\emptyset$	187, 188, 194, 195, 211
Existential path quantifier	Specifies that a temporal logic proposition must be satisfied along at least one path for a computational tree logic specification	E	29, 30, 66, 70, 71, 75

Name	Description	Symbol	Page(s)
Existential variable quantifier	Specifies that a predicate must be true for at least one value of a variable identified on the right-hand side	$\exists$	29, 186–188, 260
Finite, nonempty set	Represents a nonempty set containing a finite number of elements	$\mathbb{F}_1$	xxi, 192, 206
Future-state	Specifies that a logical proposition must eventually be *true* for a temporal logic specification	F	29, 30, 66, 70, 71, 75, 153, 154
Generalized intersection	Set operator specifying the intersection of common elements in n sets	$\bigcap_{i=1}^{n}$	187, 188
Generalized union	Set operator specifying the union of all elements in n sets	$\bigcup_{i=1}^{n}$	63, 194
Greater than	Arithmetic inequality sign specifying that a variable on the left-hand side is greater than the variable on the right-hand side	$>$	44, 244
Greater than/equal to	Arithmetic inequality sign specifying that a variable on the left-hand side is greater than or equal to the variable on the right-hand side	$\geq$	62, 193

Name	Description	Symbol	Page(s)
Hide	Specifies a set of variables on the right-hand side that are removed from a Z schema on the left-have side	$\setminus$	188, 207
In the set	Set operator specifying a variable on the left-hand side that is in a set on the right-hand side	$\in$	63, 187, 188, 194, 195, 211
Initial states	The set of initial states in the Kripke structure representation of a formal model	S_0	42
Integers	The set of all integers	$\mathbb{Z}$	62, 63
Labels	State labels in the Kripke structure representation of a formal model	L	42
Less than	Arithmetic inequality sign specifying that a variable on the left-hand side is less than the variable on the right-hand side	$<$	43, 193
Less than/equal to	Arithmetic inequality sign specifying that a variable on the left-hand side is less than or equal to the variable on the right-hand side	$\leq$	62, 75

Name	Description	Symbol	Page(s)
Logical conjunction	Boolean operator specifying that predicates on the left- and right-hand sides must be true	$\wedge$	29, 66, 71, 75, 153, 154, 168, 194–199, 211, 244, 261–273
Logical disjunction	Boolean operator specifying that one or both predicates on the left- and right-hand sides must be true	$\vee$	29, 190, 191, 194, 225, 263, 264, 267

Name	Description	Symbol	Page(s)
Logical implication	Logical connective specifying that predicates on the left-hand side valued true imply that predicates are the right-and side are also true	$\Rightarrow$	29, 32, 87, 88, 120, 150, 153, 154, 193–195, 199, 211, 244, 261–272
Logical negation	Boolean operator specifying that a predicate on the right-hand side must be false	$\neg$	29, 153, 154, 168, 199, 262–265, 267–273
Mapping	Injective function specifying a mapping of inputs to outputs in a one-to-one way	$\mapsto$	63
Model	Represents a formal model in the Kripke structure representation	$\mathcal{M}$	42

Name	Description	Symbol	Page(s)
Multiplication	Arithmetic multiplication operator when there are numbers or variables representing numbers on the left- and right-hand sides; tuple operator when there are variables representing sets on the left- and right-hand sides	$\times$	125, 166, 169, 170, 245–247
Naturals	The set of all natural numbers	$\mathbb{N}$	43
Next-state	Specifies that a logical proposition must be *true* in the next-state for a temporal logic specification	X	29, 30, 66, 71, 260, 262, 265, 266, 268, 270–272
Not equal to	Arithmetic non-equality sign specifying that a variable on the left-hand side is not equal to a variable on the right-hand side	$\neq$	29, 190
Phi	Lowercase Greek letter phi representing a formal model variable or a logical proposition	ϕ	xvi, 29, 30

Name	Description	Symbol	Page(s)
Psi	Lowercase Greek letter psi representing a formal model variable or a logical proposition	ψ	xvi, 29, 30
SAL conjunction	Logical conjunction operator in the syntax of Symbolic Analysis Laboratory (SAL)	`AND`	88, 89, 99, 101, 103, 107, 110, 112, 165, 220, 221, 243, 328, 336
SAL disjunction	Logical disjunction operator in the syntax of SAL	`OR`	112, 235, 237, 246, 247
SAL function	Specifies an input/output function in the syntax of SAL	`->`	49

Name	Description	Symbol	Page(s)
SAL guard closing	Specifies the end of a guard command in the syntax of SAL	`-->`	36–38, 46, 50, 65, 236, 237, 242, 243
SAL guard opening	Specifies the beginning of a guard command in the syntax of SAL when utilizing the initialization or transition constructs. Specifies the asynchronous composition of SAL modules when utilized outside the initialization or transition constructs	`[]`	36–38, 46, 50, 51, 65, 85, 236, 237, 242, 243, 330
SAL implication	Logical implication operator in the syntax of SAL	`=>`	51, 52, 88, 89, 111, 112, 165, 220, 221, 246, 247

Name	Description	Symbol	Page(s)
SAL in set	Specifies that the value of a variable listed on the left-hand side must be in the set of values specified on the right-hand side	`IN`	47–49, 88, 89, 105–108, 112, 242, 243, 245–247
SAL let definition	Specifies a "let" expression in the syntax of SAL. See [68] for more information	`LET`	89
SAL negation	Logical negation operator in the syntax of SAL	`NOT`	165, 166, 221
SAL not equals	Non-equality symbol in the syntax of SAL	`/=`	220
SAL turnstile	Utilized in the SAL theorem construct to specify a temporal logic specification on the right-hand side and the SAL modules being checked on the left-hand side	`|-`	45, 51, 52, 88, 89, 112, 165, 166, 246, 247
Set comprehension	Separates a variable on the left-hand side and one predicate on the right-hand side specifying its allowed values	$\mid$	63, 65

Name	Description	Symbol	Page(s)
Set formation	Separates a local variable on the left-hand side and a set of predicates on the right-hand side specifying its allowed values	$\bullet$	63, 65, 187, 190, 194, 209, 260, 268, 269, 271, 272
Set union	Set operator specifying the union of all elements in two sets on the left- and right-hand sides	$\cup$	63
States	States in the Kripke structure representation of a formal model	S	42
Synchronous composition	Specifies the synchronous composition of SAL modules	||	51, 52, 145, 330
Transition	Transitions in the Kripke structure representation of a formal model	$\rightarrow$	42
Universal path quantifier	Specifies that a temporal logic proposition must be satisfied along all paths for a computational tree logic specification	A	30

Name	Description	Symbol	Page(s)
Universal variable quantifier	Specifies that a predicate must be true for all values of a variable identified on the right-hand side	$\forall$	29, 63, 187, 188, 190, 191, 194, 209, 210

Appendix B: Tool Support Listing

Little-JIL [132] is a graphical description language and accompanying encoding environment [230] for representing hierarchical-heterarchical, normative human task behavior as a stepwise processes involving one or more agents and resources at each step. Processes can be physical or cognitive, agents can be human or computer, and resources can be physical or environmental. The analyst can specify temporal and cardinal ordering of steps as well as logical expressions controlling pre- and post-requisite step conditions. Little-JIL representations are not amenable to formal verification, but researchers have developed an automated tool that parses instantiated descriptions and generates formal task models [231]. Little-JIL processes have been employed in a number of complex human-interactive applications incorporating multiple human agents and verbal communications among them [232, 233].

ConcurTaskTrees (CTT) [66] is a diagrammatic notation for representing hierarchical, goal-oriented human task behavior. The notation includes a taxonomy of tasks, such as *user-processing tasks* involving cognitive processing of inputs from a visual display. Its formal semantics enable the analyst to specify temporal and cardinal ordering of cognitive and motor user tasks as well as other tasks that involve a device or software application (called *application tasks*). To support model checking analyses of CTT representations, researchers have developed ways of transforming task analytic diagrams to FSMs [141, 138]. One such approach translates an instantiated CTT to an intermediate abstraction called a Presentation Task Set (PTS), which is then converted to an FSM [234, 235]. This technique has been employed to support model checking analyses of recently evolving technologies such as pervasive computing environments [236] and adaptive user interfaces [237].

The HAMSTERS diagrammatic notation [238] extends CTT with semantics for representing discrete device control logic. A graphical encoding and analysis environment [142] enables translation of HAMSTERS diagrams to FSM representations of human-system interaction and analyses using a built-in model checking suite. The HAMSTERS notation has been extended to incorporate concept maps that represent the end user's declarative knowledge of a target system, as well as how it evolves during human-system interaction [134, 239].

Appendix C: Listing of Additional Encoding Techniques

C.1 HES Module Syntax for Spatial Relation Transitions Using Stoffregen's Formalism

The syntax below differs from what was shown in Chapter 6, Section 6.3.12 by replacing Boolean type values with Boolean tuples, since the translator generates SAL types `[BOOLEAN, BOOLEAN]` for Stoffregen's formalism.

```
TRANSITION [
 affordance_1 = [TRUE, TRUE] -->
  entity_1' = ...
        ...
  entity_n' = ...
 affordance_m = [TRUE, TRUE] -->
  entity_1' = ...
        ...
  entity_n' = ...
  ...
 []ELSE -->
];
```

C.2 HES Module Syntax for Initializing Human Capabilities

As mentioned in Chapter 6, Section 6.3.12, there are many ways to initialize human capabilities in a formal human-environment system model. Two examples are derived here from the "door openable" (Section 6.3 and "book placeable" affordances (Section 6.3.11).

In the "door openable" example, the analyst should reason about whether the human operator's hand is wide enough to grip the knob and whether she is strong enough to turn it. The analyst can optionally specify whether the human operator can successfully execute other 6DoF movements on the knob, such as pitching it up or down, which could be possible if, for example, the knob is poorly secured within the door. If it is only possible for her to rotate the knob, one way to specify this capability is shown below.

```
INITIALIZATION
 (FORALL x:  translate):  pHuman_sKnob.translatable[x] = false;
 (FORALL x:  position):  pHuman_sKnob.positionable[x] = false;
 (FORALL x:  orient):  pHuman_sKnob.orientable[x] =
  IF x = roll_right OR x = roll_left THEN true ELSE false ENDIF;
```

This code specifies that the human operator cannot translate or position the knob in any direction, can roll it leftward or rightward, and each movement exists independently of others.

For the "book placeable" example, the analyst should reason about whether the human operator's physical characteristics are sufficient for gripping and moving the book about its origin in any situation that is possible in the HES, Consider the book placed flat on the floor, front cover facing up. The floor does not need to be not modeled as a variable, but the analyst should reason about its texture and material to aid in determining what movements are possible. Suppose the human operator has two hands that are large enough to grip the book, but he is not strong enough to lift more than two corners off any flat surface at the same time. Friction between the floor/book and

tabletop surface/book do not restrict the human operator from pushing it in any direction in the x-y plane. One way to specify such a capability is shown below.

```
INITIALIZATION [
 (pHuman_aoBook.orientable[pitch_up] XOR pHuman_aoBook.orientable[pitch_back]
  XOR
 pHuman_aoBook.orientable[roll_left] XOR pHuman_aoBook.orientable[roll_right])
  AND
 (pHuman_aoBook.positionable[up] OR pHuman_aoBook.positionable[down] -->
  pHuman_aoBook.positionable[down] = false;
  pHuman_aoBook.positionable[back] = true;
  pHuman_aoBook.positionable[forth] = true;
  pHuman_aoBook.translatable[right] = true;
  pHuman_aoBook.translatable[left] = true;
  pHuman_aoBook.orientable[yaw_right] = true;
  pHuman_aoBook.orientable[yaw_left] = true;
  pHuman_aoBook.orientable[yaw_left] = true;
];
```

This code specifies that the human operator can position the book up or down in the z axis *and* orient it in exactly one of three ways (denoted by the SAL keyword `XOR`):

1. pitch upward
2. pitch backward
3. roll leftward
4. roll rightward

In other words, it is not possible for the human operator to position the book upward or downward without also orienting it in exactly one way, since he is not strong enough to lift more than two corners off a flat surface at once. If the book is on the floor, the upward movement(s) are possible. If the book is on the table, either upward or downward movements are possible.

C.3 Abstracting Learned Documentation Content in the Integrated Framework

In the case study of Chapter 9, the page number output of the *documentation navigation* model was utilized as an input to *signifier* and *task* models. The other technique mentioned in Section 9.1 requires an additional set of output variables to abstract learned content based on what pages have been visited. The SAL syntax for employing such a technique is encoded generally below (annotations added in italic text) and explained in the following paragraph.

```
documentation:  CONTEXT =
BEGIN
 keepPage(page:  INTEGER): INTEGER = page;
 turnPage(page:  INTEGER): INTEGER = page + 1;
 crossRef(ref :  INTEGER): INTEGER = ref;
 pages:  TYPE = {x:  INTEGER | x >= 0 AND x <= n}; where "n" is the last page
  navigation:  MODULE =
  BEGIN
   LOCAL page:  pages                           Page number is now a local variable.
   OUTPUT iSeenPage:  ARRAY pages OF BOOLEAN  Each element of this array represents
                                               whether the page has been seen.
   INITIALIZATION                              Initially, no page has been seen;
    FORALL(x:  pages):  iSeenPage[x] = FALSE; thus, all elements of iSeenPage are false
   TRANSITION [
   page = 0 -->                               From the table of contents (page-0),
    page' IN {crossRef(i),..., crossRef(j)}; the user can navigate to pages i–j
    iSeenPage'[0] = TRUE;                     and the end user has seen page-0.
   page = i -->                                 From page-i, the user can stay
    page' IN {keepPage(page), turnPage(page)}; or turn to the next page
    iSeenPage'[i] = TRUE;                        and the end user has seen page-i.
    ...
   page = n -->                               From page-n, the user can stay
    page' IN {keepPage(page), crossRef(i)}; or navigate to page-i
    iSeenPage'[i] = TRUE;                    and the end user has seen page-n.
  END;
END
```

Instead of encoding the output variable "*iPage*" (as in Chapter 9), the analyst should represent page number as a local variable, since its value is now only relevant to the *documentation navigation* model. The array of Boolean variables *iSeenPage* is a new output representing whether the end user has navigated to each page. As shown in the transition construct, each element of *iSeenPage* transitions permanently to *true* once the page has been navigated to. The element of *iSeenPage* corresponding to a page number can be leveraged in the same way as *iPage* was leveraged in the case study of Chapter 9 (as an input to *task* and *signifier* models). As an input to a *task* model, the page-number element of *iSeenPage* must be valued *true* for procedural steps to execute, indicating that the end user has seen the page containing those steps and has learned them. As an input to the *signifier* model, the page-number element of *iSeenPage* must be valued *true* for a function or meaning explained on that page to be signified through the documentation channel, indicating that the end user has seen the page containing that explanation and has learned it.

Appendix D: Chapter 4 Code Listing

D.1 SAL Model of User Manual Navigation (Section 4.3.1)

```
1  documentation: CONTEXT =
2  BEGIN
3    keepPage(page: INTEGER): INTEGER = page;
4    turnPage(page: INTEGER): INTEGER = page + 1;
5    crossRef(ref: INTEGER): INTEGER = ref;
6    navigation: MODULE =
7    BEGIN
8      OUTPUT page: {x: INTEGER | x >= 0 AND x <= 263}
9      INITIALIZATION
10      page = 0;
11      TRANSITION
12       [
13       page = 0 -->
14         page' IN {crossRef(11), crossRef(66), crossRef(101),
15           crossRef(178), crossRef(200), crossRef(202), crossRef(206),
16           crossRef(214), crossRef(225)};
17       []page = 11 -->
18         page' IN {crossRef(22), crossRef(140)};
19       []page = 66 -->
20         page' IN {keepPage(page), crossRef(93), crossRef(99)};
21       []page = 70 -->
22         page' IN {crossRef(88), crossRef(263), crossRef(22)};
23       []page = 75 -->
24         page' IN {keepPage(page), turnPage(page)};
25       []page = 101 -->
26         page' IN {crossRef(93), crossRef(99)};
27       []page = 200 -->
28         page' IN {keepPage(200), crossRef(210)};
29       []page = 202 -->
30         page' IN {keepPage(page), turnPage(page), crossRef(203), crossRef(121)};
31       []page = 203 -->
32         page' IN {keepPage(page), crossRef(121)};
33       []page = 206 -->
34         page' IN {keepPage(page), turnPage(page)};
35       []page = 207 -->
36         page' IN {keepPage(page), turnPage(page), crossRef(210)};
37       []page = 210 -->
38         page' IN {keepPage(page), crossRef(121), crossRef(214)};
39       []page = 225 -->
40         page' IN {crossRef(75), crossRef(140)};
41     ];
42   END;
43 END
```

Appendix E: Chapter 5 Code Listing

E.1 EOFM-XML Formal Task Model (Section 5.3.3)

The *ord* model is shown. The *or_seq* model was encoded by replacing the *decomposition* node on line-98 with <decomposition operator="or_seq">.

```
<?xml version="1.0" encoding="UTF-8"?>
<eofms xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="../../../../Research/git/schema/OFMr8.xsd">
 <humanoperator name="pPatient">
  <inputvariable name="iOldComponentTags" basictype="device!PartTag"/>
  <inputvariable name="iYCableToOldController" basictype="device!Connection"/>
  <inputvariable name="iNewLiBattCableToNewLiBatt" basictype="device!Connection"/>
  <inputvariable name="iNewLiBattCableToOldLiBatt" basictype="device!Connection"/>
  <inputvariable name="iNewLiBattCableToNewController" basictype="device!Connection"/>
  <inputvariable name="iOldLiBattCableToOldLiBatt" basictype="device!Connection"/>
  <inputvariable name="iOldLiBattCableToYCable" basictype="device!Connection"/>
  <inputvariable name="iOldLiBattCableToOldController" basictype="device!Connection"/>
  <inputvariable name="iPumpCableToOldAbCable" basictype="device!Connection"/>
  <inputvariable name="iPumpCableToNewController" basictype="device!Connection"/>
  <inputvariable name="iAbCableToOldController" basictype="device!Connection"/>
  <inputvariable name="iLeadBattToNewController" basictype="device!Connection"/>
  <inputvariable name="iOldLiBatteryLights" basictype="device!LiBatteryLights"/>
  <inputvariable name="iNewLiBatteryLights" basictype="device!LiBatteryLights"/>
  <inputvariable name="iPermanentlyAttachedConnector" basictype="device!
      PermAttachedConnectorStatus"/>
  <inputvariable name="iRotationCounter" basictype="device!RotationCounter"/>
  <inputvariable name="iNewControllerAlarmBatteryCap" basictype="device!AlarmBatteryCap"/>
  <inputvariable name="iOldControllerAlarmBatteryCap" basictype="device!AlarmBatteryCap"/>
  <inputvariable name="iOldComponents" basictype="device!OldComponentsLocation"/>
  <inputvariable name="iAlarm" basictype="device!Alarm"/>
  <humanaction name="hContactEmergency" behavior="autoreset"/>
  <humanaction name="hRedTagOldComponents" behavior="autoreset"/>
  <humanaction name="hSetAsideOldComponents" behavior="autoreset"/>
  <humanaction name="hDisassembleConnector" behavior="autoreset"/>
  <humanaction name="hRotateConnectorParts" behavior="autoreset"/>
  <humanaction name="hReassembleBrokenConnector" behavior="autoreset"/>
  <humanaction name="hLoosenOldControllerABCap" behavior="autoreset"/>
  <humanaction name="hTightenNewControllerABCap" behavior="autoreset"/>
  <humanaction name="hLoosenNewControllerABCap" behavior="autoreset"/>
  <humanaction name="hDiscPumpCableFromAbCable" behavior="autoreset"/>
  <humanaction name="hDiscOldAbCableFromOldController" behavior="autoreset"/>
  <humanaction name="hConNewLiBattCableToNewController" behavior="autoreset"/>
  <humanaction name="hConNewLiBattCableToNewLiBatt" behavior="autoreset"/>
  <humanaction name="hConNewLiBattCableToOldLiBatt" behavior="autoreset"/>
  <humanaction name="hDiscOldLiBattCableFromOldController" behavior="autoreset"/>
  <humanaction name="hDiscOldLiBattCableFromYCable" behavior="autoreset"/>
  <humanaction name="hDiscOldLiBattCableFromOldLiBatt" behavior="autoreset"/>
  <humanaction name="hDiscNewLiBattCableFromOldLiBatt" behavior="autoreset"/>
  <humanaction name="hConPumpCableToNewController" behavior="autoreset"/>
  <humanaction name="hDiscNewLiBattCableFromNewLiBatt" behavior="autoreset"/>
  <humanaction name="hConLeadBattToNewController" behavior="autoreset"/>
  <humanaction name="hDiscLeadBattFromNewController" behavior="autoreset"/>
  <humanaction name="hDiscNewLiBattCableFromNewController" behavior="autoreset"/>
  <humanaction name="hDepressBlackButtonOnNewLiBatt" behavior="autoreset"/>
  <humanaction name="hDepressBlackButtonOnOldLiBatt" behavior="autoreset"/>
  <eofm> <!-- follows page 14 of patient handbook -->
   <activity name="aRespondToPumpStoppedAlarm">
    <precondition>iAlarm = device!PumpStopped</precondition>
    <completioncondition>iAlarm /= device!PumpStopped</completioncondition>
    <decomposition operator="ord">
     <!-- begin step 1 -->
     <activity name="aStep1aDiscOldParts">
      <decomposition operator="ord">
       <activity name="aDiscPumpCableFromAbCable">
        <precondition>iPumpCableToOldAbCable = device!Connected</precondition>
        <completioncondition>iPumpCableToOldAbCable = device!Disconnected</completioncondition>
        <decomposition operator="ord">
         <action humanaction="hDiscPumpCableFromAbCable"/>
        </decomposition>
       </activity>
       <activity name="aSetAsideOldComponents">
```

```
66        <precondition>iOldComponents = device!AtHand</precondition>
67        <completioncondition>iOldComponents = device!SetAside</completioncondition>
68        <decomposition operator="ord">
69         <action humanaction="hSetAsideOldComponents"/>
70        </decomposition>
71       </activity>
72       <activity name="aDiscOldLiBattery">
73        <decomposition operator="xor">
74         <activity name="aDiscOldBattCableFromYCable">
75          <precondition>iOldLiBattCableToYCable = device!Connected</precondition>
76          <decomposition operator="or_seq">
77           <activity name="aDiscOldLiBattFromOldBattCable">
78            <precondition>iOldLiBattCableToOldLiBatt = device!Connected</precondition>
79            <completioncondition>iOldLiBattCableToOldLiBatt = device!Disconnected</
                  completioncondition>
80            <decomposition operator="ord">
81             <action humanaction="hDiscOldLiBattCableFromOldLiBatt"/>
82            </decomposition>
83           </activity>
84           <activity name="aDiscBattCableFromYCable">
85            <completioncondition>iOldLiBattCableToYCable = device!Disconnected</
                  completioncondition>
86            <decomposition operator="ord">
87             <action humanaction="hDiscOldLiBattCableFromYCable"/>
88            </decomposition>
89           </activity>
90          </decomposition>
91         </activity>
92         <activity name="aDiscOldBattCableFromOldController">
93          <precondition>iOldLiBattCableToOldController = device!Connected</precondition>
94          <decomposition operator="or_seq">
95           <activitylink link="aDiscOldLiBattFromOldBattCable"/>
96           <activity name="aDiscBattCableFromController">
97            <completioncondition>iOldLiBattCableToOldController = device!Disconnected</
                  completioncondition>
98            <decomposition operator="ord">
99             <action humanaction="hDiscOldLiBattCableFromOldController"/>
100           </decomposition>
101          </activity>
102         </decomposition>
103        </activity>
104       </decomposition>
105      </activity>
106      <activity name="aSilenceAlarmOnOldController">
107       <precondition>iOldControllerAlarmBatteryCap = device!Tightened</precondition>
108       <completioncondition>iOldControllerAlarmBatteryCap = device!Loosened</completioncondition
              >
109       <decomposition operator="ord">
110        <action humanaction="hLoosenOldControllerABCap"/>
111       </decomposition>
112      </activity>
113     </decomposition>
114    </activity>
115    <activity name="aStep1bConNewController">
116     <decomposition operator="ord">
117      <activity name="aConPumptoNewController">
118       <precondition>iPumpCableToNewController = device!Disconnected and iPumpCableToOldAbCable
              = device!Disconnected</precondition>
119       <completioncondition>iPumpCableToNewController = device!Connected</completioncondition>
120       <decomposition operator="ord">
121        <action humanaction="hConPumpCableToNewController"/>
122       </decomposition>
123      </activity>
124      <activity name="aActivateAlarmOnNewController">
125       <precondition>iNewControllerAlarmBatteryCap = device!Loosened</precondition>
126       <completioncondition>iNewControllerAlarmBatteryCap = device!Tightened</
              completioncondition>
127       <decomposition operator="ord">
128        <action humanaction="hTightenNewControllerABCap"/>
129       </decomposition>
130      </activity>
131     </decomposition>
132    </activity>
133    <activity name="aStep1cFullyChargedLiBatt">
134     <decomposition operator="ord">
135      <activity name="aConNewBattCable">
136       <precondition>iNewLiBattCableToNewController = device!Disconnected</precondition>
137       <completioncondition>iNewLiBattCableToNewController = device!Connected</
              completioncondition>
138       <decomposition operator="ord">
139        <action humanaction="hConNewLiBattCableToNewController"/>
```

```
140      </decomposition>
141     </activity>
142     <activity name="aConnectFullyChargedLiBattery">
143      <precondition>iNewLiBattCableToNewController = device!Connected</precondition>
144      <decomposition operator="xor">
145       <activity name="aConnectNewLiBattToNewLiBattCable">
146        <completioncondition>iNewLiBattCableToNewLiBatt = device!Connected</completioncondition>
147        <decomposition operator="ord">
148         <action humanaction="hConNewLiBattCableToNewLiBatt"/>
149        </decomposition>
150       </activity>
151       <activity name="aConnectOldLiBattToNewLiBattCable">
152        <completioncondition>iNewLiBattCableToOldLiBatt = device!Connected</completioncondition>
153        <decomposition operator="ord">
154         <action humanaction="hConNewLiBattCableToOldLiBatt"/>
155        </decomposition>
156       </activity>
157      </decomposition>
158     </activity>
159    </decomposition>
160   </activity>
161   <!-- end step 1 -->
162   <!-- begin step 2 -->
163   <activity name="aStep2CallEmergencyNumber">
164    <decomposition operator="ord">
165     <action humanaction="hContactEmergency"/>
166    </decomposition>
167   </activity>
168   <!-- end step 2 -->
169   <!-- begin step 3 -->
170   <activity name="aStep3RedTagOldComponents">
171    <precondition>iOldComponentTags = device!notRedTagged</precondition>
172    <completioncondition>iOldComponentTags = device!redTagged</completioncondition>
173    <decomposition operator="ord">
174     <action humanaction="hRedTagOldComponents"/>
175    </decomposition>
176   </activity>
177   <!-- end step 3 -->
178   <!-- begin step 4 -->
179   <activity name="aStep4FixConnector">
180    <precondition>iPermanentlyAttachedConnector = device!Broken</precondition>
181    <decomposition operator="ord">
182     <activity name="aReassembleConnector">
183      <completioncondition>iPermanentlyAttachedConnector = device!Assembled</completioncondition>
184      <decomposition operator="ord">
185       <action humanaction="hReassembleBrokenConnector"/>
186      </decomposition>
187     </activity>
188     <activity name="aTryRotatingParts">
189      <precondition>iRotationCounter = 0</precondition>
190      <repeatcondition>iRotationCounter &lt; 3</repeatcondition>
191      <completioncondition>iRotationCounter = 3</completioncondition>
192      <decomposition operator="ord">
193       <action humanaction="hDisassembleConnector"/>
194       <action humanaction="hRotateConnectorParts"/>
195       <action humanaction="hReassembleBrokenConnector"/>
196      </decomposition>
197     </activity>
198    </decomposition>
199   </activity>
200   <!-- end step 4 -->
201   <!-- begin step 5i-->
202   <activity name="aStep5aChangeBatts">
203    <decomposition operator="ord">
204     <activity name="aSwitchLiBatteries">
205      <decomposition operator="xor">
206       <activity name="aSwitchFromOldToNew">
207        <precondition>iNewLiBattCableToOldLiBatt = device!Connected</precondition>
208        <completioncondition>iNewLiBattCableToNewLiBatt = device!Connected</completioncondition>
209        <decomposition operator="ord">
210         <action humanaction="hDiscNewLiBattCableFromOldLiBatt"/>
211         <action humanaction="hConNewLiBattCableToNewLiBatt"/>
212        </decomposition>
213       </activity>
214       <activity name="aSwitchFromNewToOld">
215        <precondition>iNewLiBattCableToOldLiBatt = device!Connected</precondition>
```

```
216           <completioncondition>iNewLiBattCableToNewLiBatt = device!Connected</completioncondition
                  >
217           <decomposition operator="ord">
218            <action humanaction="hDiscNewLiBattCableFromNewLiBatt"/>
219            <action humanaction="hConNewLiBattCableToOldLiBatt"/>
220           </decomposition>
221          </activity>
222         </decomposition>
223        </activity>
224       </decomposition>
225      </activity>
226      <!-- end step 5i -->
227      <activity name="aStep5bCheckAndSwitchToLead">
228       <decomposition operator="ord">
229        <activity name="aCheckLiBatteryLevels">
230          <decomposition operator="and_seq">
231          <action humanaction="hDepressBlackButtonOnNewLiBatt"/>
232          <action humanaction="hDepressBlackButtonOnOldLiBatt"/>
233         </decomposition>
234        </activity>
235        <activity name="aSwitchToLeadOrKeepLiBatt">
236         <decomposition operator="ord">
237          <activity name="aSwitchToLeadBatt">
238           <precondition>iNewLiBatteryLights = 0 AND iOldLiBatteryLights = 0</precondition>
239           <completioncondition>iLeadBattToNewController = device!Connected</completioncondition>
240           <decomposition operator="ord">
241            <action humanaction="hDiscNewLiBattCableFromNewController"/>
242            <action humanaction="hConLeadBattToNewController"/>
243           </decomposition>
244          </activity>
245         </decomposition>
246        </activity>
247       </decomposition>
248      </activity>
249      <!-- end step 5 -->
250      <!-- begin step 6i -->
251      <activity name="aStep6aBreakCircuit">
252       <decomposition operator="xor">
253        <activity name="aDiscLeadBatt">
254         <precondition>iLeadBattToNewController = device!Connected</precondition>
255         <completioncondition>iLeadBattToNewController = device!Disconnected</completioncondition>
256         <decomposition operator="ord">
257          <action humanaction="hDiscLeadBattFromNewController"/>
258         </decomposition>
259        </activity>
260        <activity name="aDiscLiBatt">
261         <precondition>iNewLiBattCableToNewController = device!Connected</precondition>
262         <completioncondition>iLeadBattToNewController = device!Disconnected</completioncondition>
263         <decomposition operator="ord">
264          <action humanaction="hDiscNewLiBattCableFromNewController"/>
265         </decomposition>
266        </activity>
267       </decomposition>
268      </activity>
269      <!-- end step 6i -->
270      <!-- begin step 6ii -->
271      <activity name="aStep6bSilenceNewController">
272       <precondition>iNewControllerAlarmBatteryCap = device!Tightened</precondition>
273       <completioncondition>iNewControllerAlarmBatteryCap = device!Loosened</completioncondition>
274       <decomposition operator="ord">
275        <action humanaction="hLoosenNewControllerABCap"/>
276       </decomposition>
277      </activity>
278      <!-- end step 6 -->
279     </decomposition>
280    </activity>
281   </eofm>
282  </humanoperator>
283 </eofms>
```

E.2 SAL Model of Human-Device Interaction (Section 5.3.5)

```
1 device: CONTEXT =
2 BEGIN
3 RotationCounter: TYPE = [0..3];
4 OldComponentsLocation: TYPE = {AtHand, SetAside};
5 Connection: TYPE = {Connected, Disconnected};
```

```
6   PartTag: TYPE = {redTagged, notRedTagged};
7   PermAttachedConnectorStatus: TYPE = {Broken, Assembled};
8   AlarmBatteryCap: TYPE = {Loosened, Tightened};
9   Alarm: TYPE = {PumpStopped, NoAlarm};
10  LiBatteryLights: TYPE = [0..5];
11  LeadBattLight: TYPE = [0..1];
12    HDI: MODULE =
13    BEGIN
14      INPUT hRedTagOldComponents: BOOLEAN
15      INPUT hSetAsideOldComponents: BOOLEAN
16      INPUT hRotateConnectorParts: BOOLEAN
17      INPUT hDisassembleConnector: BOOLEAN
18      INPUT hReassembleBrokenConnector: BOOLEAN
19      INPUT hLoosenOldControllerABCap: BOOLEAN
20      INPUT hTightenNewControllerABCap: BOOLEAN
21      INPUT hLoosenNewControllerABCap: BOOLEAN
22      INPUT hDiscPumpCableFromAbCable: BOOLEAN
23      INPUT hConNewLiBattCableToNewController: BOOLEAN
24      INPUT hConNewLiBattCableToNewLiBatt: BOOLEAN
25      INPUT hConNewLiBattCableToOldLiBatt: BOOLEAN
26      INPUT hDiscOldLiBattCableFromOldController: BOOLEAN
27      INPUT hDiscOldLiBattCableFromYCable: BOOLEAN
28      INPUT hDiscOldLiBattCableFromOldLiBatt: BOOLEAN
29      INPUT hConOldLiBattToNewLiBattCable: BOOLEAN
30      INPUT hConPumpCableToNewController: BOOLEAN
31      INPUT hDiscNewLiBattCableFromNewLiBatt: BOOLEAN
32      INPUT hDiscNewLiBattCableFromOldLiBatt: BOOLEAN
33      INPUT hConLeadBattToNewController: BOOLEAN
34      INPUT hDiscNewLiBattCableFromNewController: BOOLEAN
35      INPUT hDiscLeadBattFromNewController: BOOLEAN
36      INPUT hDepressBlackButtonOnNewLiBatt: BOOLEAN
37      INPUT hDepressBlackButtonOnOldLiBatt: BOOLEAN
38      INPUT submitted: BOOLEAN
39      OUTPUT iOldLiBatteryLights: LiBatteryLights
40      OUTPUT iNewLiBatteryLights: LiBatteryLights
41      OUTPUT iOldComponentTags: PartTag
42      OUTPUT iYCableToOldController: Connection
43      OUTPUT iNewLiBattCableToNewLiBatt: Connection
44      OUTPUT iNewLiBattCableToOldLiBatt: Connection
45      OUTPUT iNewLiBattCableToNewController: Connection
46      OUTPUT iOldLiBattCableToOldLiBatt: Connection
47      OUTPUT iOldLiBattCableToYCable: Connection
48      OUTPUT iOldLiBattCableToOldController: Connection
49      OUTPUT iLeadBattToYCable: Connection
50      OUTPUT iLeadBattToOldController: Connection
51      OUTPUT iPumpCableToOldAbCable: Connection
52      OUTPUT iPumpCableToOldController: Connection
53      OUTPUT iPumpCableToNewController: Connection
54      OUTPUT iAbCableToOldController: Connection
55      OUTPUT iLeadBatteryLight: LeadBattLight
56      OUTPUT iLeadBattToNewController: Connection
57      OUTPUT iPermanentlyAttachedConnector: PermAttachedConnectorStatus
58      OUTPUT iRotationCounter: RotationCounter
59      OUTPUT iNewControllerAlarmBatteryCap: AlarmBatteryCap
60      OUTPUT iOldControllerAlarmBatteryCap: AlarmBatteryCap
61      OUTPUT iOldComponents: OldComponentsLocation
62      OUTPUT functional: BOOLEAN
63      OUTPUT ready: BOOLEAN
64
65      INITIALIZATION
66        ready = FALSE;
67        iYCableToOldController         IN {Connected, Disconnected};
68        iOldLiBattCableToOldController IN IF iYCableToOldController = Connected
69                                             THEN {Disconnected}
70                                             ELSE {Connected, Disconnected}
71                                           ENDIF;
72        iLeadBattToOldController       = IF iYCableToOldController = Connected
73                                             THEN Disconnected
74                                          ELSIF iOldLiBattCableToOldController = Connected
75                                             THEN Disconnected
76                                             ELSE Connected
77                                         ENDIF;
78        iLeadBattToYCable              = IF (iOldLiBattCableToYCable = Disconnected AND
             iYCableToOldController = Connected)
79                                             THEN Connected
80                                             ELSE Disconnected
81                                          ENDIF;
82        iOldLiBattCableToYCable        IN IF iYCableToOldController = Connected
83                                             THEN {Connected, Disconnected}
84                                             ELSE {Disconnected}
85                                           ENDIF;
```

```
86       iOldLiBattCableToOldLiBatt      = IF iOldLiBattCableToOldController = Connected OR (
             iYCableToOldController = Connected AND not(iLeadBattToYCable = Connected))
87                                             THEN Connected
88                                             ELSE Disconnected
89                                          ENDIF;
90
91       iPumpCableToOldAbCable             IN {Connected, Disconnected};
92       iPumpCableToOldController      = IF iPumpCableToOldAbCable = Connected
93                                             THEN Disconnected
94                                             ELSE Connected
95                                          ENDIF;
96       iAbCableToOldController        = IF iPumpCableToOldController = Connected
97                                             THEN Disconnected
98                                             ELSE Connected
99                                          ENDIF;
100
101       iPermanentlyAttachedConnector  IN {Broken, Assembled};
102       iOldComponents                 = AtHand;
103       iRotationCounter               = 0;
104       iNewLiBattCableToNewLiBatt     = Disconnected;
105       iNewLiBattCableToNewController = Disconnected;
106       iLeadBattToNewController       = Disconnected;
107       iPumpCableToNewController      = Disconnected;
108
109      %% Red tags
110      iOldComponentTags = notRedTagged;
111
112      %% Alarm battery caps
113      iNewControllerAlarmBatteryCap = Loosened;
114      iOldControllerAlarmBatteryCap = Tightened;
115
116      %% Battery lights
117      iOldLiBatteryLights = 0;
118      iNewLiBatteryLights = 0;
119      iLeadBatteryLight IN IF iLeadBattToOldController = Connected THEN {0, 1} ELSE {0} ENDIF;
120
121    DEFINITION
122      functional =
123        IF ((iPumpCableToOldAbCable = Connected AND iAbCableToOldController = Connected) OR
124              (iPumpCableToOldController = Connected)) AND
125             ((iLeadBattToYCable  = Connected AND iYCableToOldController = Connected AND
                 iLeadBatteryLight = 1) OR
126              (iOldLiBattCableToYCable = Connected AND iYCableToOldController = Connected AND
                  iLeadBatteryLight = 1) OR
127              (iLeadBattToOldController = Connected  AND iLeadBatteryLight = 1) OR
128              (iOldLiBattCableToOldController = Connected AND iOldLiBattCableToOldLiBatt =
                  Connected)) AND
129               iPermanentlyAttachedConnector = Assembled
130          THEN TRUE
131        ELSIF
132             iPumpCableToNewController = Connected AND
133             iPermanentlyAttachedConnector = Assembled AND
134            (iLeadBattToNewController = Connected OR
135            (iNewLiBattCableToNewLiBatt = Connected AND iNewLiBattCableToNewController =
                Connected AND iNewLiBatteryLights > 0))
136          THEN TRUE
137        ELSE FALSE ENDIF;
138
139    TRANSITION [
140        NOT (ready OR submitted) -->
141          ready' = TRUE;
142
143      []hDiscPumpCableFromAbCable and ready AND submitted -->
144        iPumpCableToOldAbCable' = Disconnected;
145        ready' = FALSE;
146
147      []hSetAsideOldComponents and ready AND submitted -->
148        iOldComponents' = SetAside;
149        ready' = FALSE;
150
151      []hDiscOldLiBattCableFromYCable and ready AND submitted -->
152        iOldLiBattCableToYCable' = Disconnected;
153        ready' = FALSE;
154
155      []hDiscOldLiBattCableFromOldController and ready AND submitted -->
156        iOldLiBattCableToOldController' = Disconnected;
157        ready' = FALSE;
158
159      []hLoosenOldControllerABCap and ready AND submitted -->
160        iOldControllerAlarmBatteryCap' = Loosened;
161        ready' = FALSE;
```

```
162
163         []hConPumpCableToNewController and ready AND submitted -->
164           iPumpCableToNewController' = Connected;
165           ready' = FALSE;
166
167         []hTightenNewControllerABCap and ready AND submitted -->
168           iNewControllerAlarmBatteryCap' = Tightened;
169           ready' = FALSE;
170
171         []hConNewLiBattCableToNewController and ready AND submitted -->
172           iNewLiBattCableToNewController' = Connected;
173           ready' = FALSE;
174
175         []hConNewLiBattCableToNewLiBatt and ready AND submitted -->
176           iNewLiBattCableToNewLiBatt' = Connected;
177           ready' = FALSE;
178
179         []hConNewLiBattCableToOldLiBatt and ready AND submitted -->
180           iNewLiBattCableToNewLiBatt' = Connected;
181           ready' = FALSE;
182
183         []hRedTagOldComponents and ready AND submitted -->
184           iOldComponentTags' = redTagged;
185           ready' = FALSE;
186
187         []hReassembleBrokenConnector and ready AND submitted -->
188           iPermanentlyAttachedConnector' = Assembled;
189           ready' = FALSE;
190
191         []hDisassembleConnector and ready AND submitted -->
192           iPermanentlyAttachedConnector' = Broken;
193           ready' = FALSE;
194         []hRotateConnectorParts and iRotationCounter < 3 and ready AND submitted -->
195           iRotationCounter' = iRotationCounter + 1;
196           ready' = FALSE;
197
198         []hDiscNewLiBattCableFromNewLiBatt and ready AND submitted -->
199           iNewLiBattCableToNewLiBatt' = Disconnected;
200           ready' = FALSE;
201
202         []hDiscOldLiBattCableFromOldLiBatt and ready AND submitted -->
203           iOldLiBattCableToOldLiBatt' = Disconnected;
204           ready' = FALSE;
205
206         []hDiscNewLiBattCableFromOldLiBatt and ready AND submitted -->
207           iNewLiBattCableToOldLiBatt' = Disconnected;
208           ready' = FALSE;
209
210         []hConNewLiBattCableToOldLiBatt and ready AND submitted -->
211           iNewLiBattCableToOldLiBatt' = Connected;
212           ready' = FALSE;
213
214         []hDepressBlackButtonOnNewLiBatt and ready AND submitted -->
215           iNewLiBatteryLights' IN {0, 1, 2, 3, 4, 5};
216           ready' = FALSE;
217         []hDepressBlackButtonOnOldLiBatt and ready AND submitted -->
218           iOldLiBatteryLights' IN {0, 1, 2, 3, 4, 5};
219           ready' = FALSE;
220
221         []hDiscNewLiBattCableFromNewController and ready AND submitted -->
222           iNewLiBattCableToNewController' = Disconnected;
223           ready' = FALSE;
224
225         []hConLeadBattToNewController and ready AND submitted -->
226           iLeadBattToNewController' = Connected;
227           iLeadBatteryLight' IN {0, 1};
228           ready' = FALSE;
229
230         []hDiscLeadBattFromNewController and ready AND submitted -->
231           iLeadBattToNewController' = Disconnected;
232           iLeadBatteryLight' = 0;
233           ready' = FALSE;
234
235         []hLoosenNewControllerABCap and ready and submitted -->
236           iNewControllerAlarmBatteryCap' = Loosened;
237           ready' = FALSE;
238
239           []ELSE -->
240           ready' = IF (ready AND submitted)
241             THEN FALSE
242             ELSE ready
```

```
243 ||          ENDIF;
244 ||        ];
245 ||      END;
246 || END
```

Appendix F: Chapter 6 Code Listing

F.1 XML Code

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <hes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xsi:noNamespaceSchemaLocation="../../schema/cavemen.xsd">
4      <modelobject name="mPulseGenerator">
5          <atomicobject name="aoLVPort"/>
6          <atomicobject name="aoRAPort"/>
7          <atomicobject name="aoRVPort"/>
8          <atomicobject name="aoLVSetScrew"/>
9          <atomicobject name="aoRASetScrew"/>
10         <atomicobject name="aoRVSetScrew"/>
11     </modelobject>
12     <atomicobject name="aoLVLeadProximalTip"/>
13     <atomicobject name="aoRALeadProximalTip"/>
14     <atomicobject name="aoRVLeadProximalTip"/>
15     <affordance        name="LVLeadConnectableToRVPort" formalism="greeno">
16         <humanoperator name="pSurgeon">
17             <relation topology="disjoint_to" direction="front_of"
18                                              associate="aoRVPort"/>
19             <relation topology="disjoint_to" associate="aoLVLeadProximalTip"/>
20             <component   name="mPulseGenerator">
21                 <ability name="MovePulseGenerator">
22                     <orientable pitch_back="true" pitch_forth="true"
23                         yaw_left="true"   yaw_right="true"/>
24                     <translatable     left="true"       right="true"/>
25                     <positionable     back="true"       forth="true"
26                         up="true"        down="true"/>
27                 </ability>
28             </component>
29             <atomcomponent     name="aoLVLeadProximalTip">
30                 <relation condition="not"      topology="covering"
31                           direction="back_of" associate="aoLVPort"/>
32                 <relation condition="not"      topology="covering"
33                            direction="back_of" associate="aoRVPort"/>
34                 <relation condition="not"      topology="covering"
35                           direction="back_of" associate="aoRAPort"/>
36                 <ability              name="MoveLVProximalTip">
37                     <orientable pitch_back="true" pitch_forth="true"
38                                   yaw_left="true"   yaw_right="true"/>
39                     <translatable     left="true"       right="true"/>
40                     <positionable     back="true"       forth="true"
41                                         up="true"        down="true"/>
42                 </ability>
43             </atomcomponent>
44             <atomcomponent    name="aoRVSetScrew">
45                 <relation topology="touching" direction="right_of"
46                                               associate="mPulseGenerator"/>
47             </atomcomponent>
48             <atomcomponent    name="aoRALeadProximalTip">
49                 <relation topology="disjoint_to" associate="aoRVPort"/>
50             </atomcomponent>
51             <atomcomponent    name="aoRVLeadProximalTip">
52                 <relation topology="disjoint_to" associate="aoRVPort"/>
53             </atomcomponent>
54         </humanoperator>
55     </affordance>
56     <affordance        name="LVLeadConnectableToLVPort" formalism="greeno">
57         <humanoperator name="pSurgeon">
58             <relation topology="disjoint_to" direction="front_of"
59                                              associate="aoLVPort"/>
60             <relation topology="disjoint_to" associate="aoLVLeadProximalTip"/>
61             <component   name="mPulseGenerator">
62                 <ability name="MovePulseGenerator">
63                     <orientable pitch_back="true" pitch_forth="true"
64                                   yaw_left="true"   yaw_right="true"/>
65                     <translatable     left="true"       right="true"/>
66                     <positionable     back="true"       forth="true"
67                                         up="true"        down="true"/>
68                 </ability>
69             </component>
```

```
70              <atomcomponent     name="aoLVLeadProximalTip">
71                  <relation condition="not"      topology="covering"
72                            direction="back_of" associate="aoLVPort"/>
73                  <relation condition="not"      topology="covering"
74                            direction="back_of" associate="aoRVPort"/>
75                  <relation condition="not"      topology="covering"
76                            direction="back_of" associate="aoRAPort"/>
77                  <ability               name="MoveLVProximalTip">
78                      <orientable pitch_back="true" pitch_forth="true"
79                                  yaw_left="true"   yaw_right="true"/>
80                      <translatable     left="true"       right="true"/>
81                      <positionable     back="true"       forth="true"
82                                          up="true"        down="true"/>
83                  </ability>
84              </atomcomponent>
85              <atomcomponent    name="aoLVSetScrew">
86                  <relation topology="touching" direction="right_of"
87                                                associate="mPulseGenerator"/>
88              </atomcomponent>
89              <atomcomponent    name="aoRALeadProximalTip">
90                  <relation topology="disjoint_to" associate="aoLVPort"/>
91              </atomcomponent>
92              <atomcomponent    name="aoRVLeadProximalTip">
93                  <relation topology="disjoint_to" associate="aoLVPort"/>
94              </atomcomponent>
95          </humanoperator>
96      </affordance>
97      <affordance        name="RVLeadConnectableToRVPort" formalism="greeno">
98          <humanoperator name="pSurgeon">
99              <relation topology="disjoint_to" direction="front_of"
100                                              associate="aoRVPort"/>
101             <relation topology="disjoint_to" associate="aoRVLeadProximalTip"/>
102             <component   name="mPulseGenerator">
103                 <ability name="MovePulseGenerator">
104                     <orientable pitch_back="true" pitch_forth="true"
105                                 yaw_left="true"   yaw_right="true"/>
106                     <translatable     left="true"       right="true"/>
107                     <positionable     back="true"       forth="true"
108                                         up="true"        down="true"/>
109                 </ability>
110             </component>
111             <atomcomponent     name="aoRVLeadProximalTip">
112                 <relation condition="not"      topology="covering"
113                           direction="back_of" associate="aoLVPort"/>
114                 <relation condition="not"      topology="covering"
115                           direction="back_of" associate="aoRVPort"/>
116                 <relation condition="not"      topology="covering"
117                           direction="back_of" associate="aoRAPort"/>
118                 <ability               name="MoveRVProximalTip">
119                     <orientable pitch_back="true" pitch_forth="true"
120                                 yaw_left="true"   yaw_right="true"/>
121                     <translatable     left="true"       right="true"/>
122                     <positionable     back="true"       forth="true"
123                                         up="true"        down="true"/>
124                 </ability>
125             </atomcomponent>
126             <atomcomponent    name="aoRVSetScrew">
127                 <relation topology="touching" direction="right_of"
128                                               associate="mPulseGenerator"/>
129             </atomcomponent>
130             <atomcomponent    name="aoRALeadProximalTip">
131                 <relation topology="disjoint_to" associate="aoRVPort"/>
132             </atomcomponent>
133             <atomcomponent    name="aoLVLeadProximalTip">
134                 <relation topology="disjoint_to" associate="aoRVPort"/>
135             </atomcomponent>
136         </humanoperator>
137     </affordance>
138 </hes>
```

F.2 SAL Code

```
1 cavemen: CONTEXT =
2  BEGIN
3 position: TYPE = {up, down, back, forth};
4 translate: TYPE = {left, right};
5 orient: TYPE = {pitch_back, pitch_forth, yaw_left, yaw_right, roll_left, roll_right};
6 abilities: TYPE = [#positionable: ARRAY position OF BOOLEAN, translatable: ARRAY translate OF
      BOOLEAN, orientable: ARRAY orient OF BOOLEAN#];
```

```
7  topological: TYPE = {disjoint_to, touching, covering, overlapping};
8  directional: TYPE = {left_of, right_of, top_of, bottom_of, front_of, back_of};
9  relations: TYPE = ARRAY directional OF topological;
10 pSurgeon_rels: TYPE = [#aoLVPort: relations, aoLVLeadProximalTip: relations, aoRVPort: relations,
       aoRVLeadProximalTip: relations#];
11 aoLVLeadProximalTip_rels: TYPE = [#aoLVPort: relations, aoRVPort: relations, aoRAPort: relations
       #];
12 aoLVSetScrew_rels: TYPE = [#mPulseGenerator: relations#];
13 aoRALeadProximalTip_rels: TYPE = [#aoRVPort: relations, aoLVPort: relations#];
14 aoRVLeadProximalTip_rels: TYPE = [#aoRVPort: relations, aoLVPort: relations, aoRAPort: relations
       #];
15 aoRVSetScrew_rels: TYPE = [#mPulseGenerator: relations#];
16 % action by pSurgeon => good effects in situation (for LVLeadConnectableToRVPort )
17 % action by pSurgeon => good effects in situation (for LVLeadConnectableToLVPort )
18 % action by pSurgeon => good effects in situation (for RVLeadConnectableToRVPort )
19 affordance: MODULE =
20  BEGIN
21   INPUT pSurgeon: pSurgeon_rels
22   INPUT aoLVLeadProximalTip: aoLVLeadProximalTip_rels
23   INPUT aoLVSetScrew: aoLVSetScrew_rels
24   INPUT aoRALeadProximalTip: aoRALeadProximalTip_rels
25   INPUT aoRVLeadProximalTip: aoRVLeadProximalTip_rels
26   INPUT aoRVSetScrew: aoRVSetScrew_rels
27   INPUT pSurgeon_mPulseGenerator: abilities
28   INPUT pSurgeon_aoLVLeadProximalTip: abilities
29   INPUT pSurgeon_aoRVLeadProximalTip: abilities
30   OUTPUT LVLeadConnectableToRVPort: BOOLEAN
31   OUTPUT LVLeadConnectableToLVPort: BOOLEAN
32   OUTPUT RVLeadConnectableToRVPort: BOOLEAN
33    DEFINITION
34   LVLeadConnectableToRVPort =
35     pSurgeon.aoLVPort[front_of] = disjoint_to AND
36     FORALL(x: directional): pSurgeon.aoLVLeadProximalTip[x] = disjoint_to AND
37     aoLVLeadProximalTip.aoLVPort[back_of] /= covering AND
38     aoLVLeadProximalTip.aoRVPort[back_of] /= covering AND
39     aoLVLeadProximalTip.aoRAPort[back_of] /= covering AND
40     aoLVSetScrew.mPulseGenerator[right_of] = touching AND
41     FORALL(x: directional): aoRALeadProximalTip.aoRVPort[x] = disjoint_to AND
42     FORALL(x: directional): aoRVLeadProximalTip.aoRVPort[x] = disjoint_to AND
43     pSurgeon_mPulseGenerator.orientable[pitch_back] = TRUE AND
44     pSurgeon_mPulseGenerator.orientable[pitch_forth] = TRUE AND
45     pSurgeon_mPulseGenerator.orientable[yaw_left] = TRUE AND
46     pSurgeon_mPulseGenerator.orientable[yaw_right] = TRUE AND
47     FORALL(x: translate): pSurgeon_mPulseGenerator.translatable[x] = TRUE AND
48     FORALL(x: position): pSurgeon_mPulseGenerator.positionable[x] = TRUE AND
49     pSurgeon_aoLVLeadProximalTip.orientable[pitch_back] = TRUE AND
50     pSurgeon_aoLVLeadProximalTip.orientable[pitch_forth] = TRUE AND
51     pSurgeon_aoLVLeadProximalTip.orientable[yaw_left] = TRUE AND
52     pSurgeon_aoLVLeadProximalTip.orientable[yaw_right] = TRUE AND
53     FORALL(x: translate): pSurgeon_aoLVLeadProximalTip.translatable[x] = TRUE AND
54     FORALL(x: position): pSurgeon_aoLVLeadProximalTip.positionable[x] = TRUE;
55   LVLeadConnectableToLVPort =
56     pSurgeon.aoLVPort[front_of] = disjoint_to AND
57     FORALL(x: directional): pSurgeon.aoLVLeadProximalTip[x] = disjoint_to AND
58     aoLVLeadProximalTip.aoLVPort[back_of] /= covering AND
59     aoLVLeadProximalTip.aoRVPort[back_of] /= covering AND
60     aoLVLeadProximalTip.aoRAPort[back_of] /= covering AND
61     aoLVSetScrew.mPulseGenerator[right_of] = touching AND
62     FORALL(x: directional): aoRALeadProximalTip.aoLVPort[x] = disjoint_to AND
63     FORALL(x: directional): aoRVLeadProximalTip.aoLVPort[x] = disjoint_to AND
64     pSurgeon_mPulseGenerator.orientable[pitch_back] = TRUE AND
65     pSurgeon_mPulseGenerator.orientable[pitch_forth] = TRUE AND
66     pSurgeon_mPulseGenerator.orientable[yaw_left] = TRUE AND
67     pSurgeon_mPulseGenerator.orientable[yaw_right] = TRUE AND
68     FORALL(x: translate): pSurgeon_mPulseGenerator.translatable[x] = TRUE AND
69     FORALL(x: position): pSurgeon_mPulseGenerator.positionable[x] = TRUE AND
70     pSurgeon_aoLVLeadProximalTip.orientable[pitch_back] = TRUE AND
71     pSurgeon_aoLVLeadProximalTip.orientable[pitch_forth] = TRUE AND
72     pSurgeon_aoLVLeadProximalTip.orientable[yaw_left] = TRUE AND
73     pSurgeon_aoLVLeadProximalTip.orientable[yaw_right] = TRUE AND
74     FORALL(x: translate): pSurgeon_aoLVLeadProximalTip.translatable[x] = TRUE AND
75     FORALL(x: position): pSurgeon_aoLVLeadProximalTip.positionable[x] = TRUE;
76   RVLeadConnectableToRVPort =
77     pSurgeon.aoRVPort[front_of] = disjoint_to AND
78     FORALL(x: directional): pSurgeon.aoRVLeadProximalTip[x] = disjoint_to AND
79     aoRVLeadProximalTip.aoLVPort[back_of] /= covering AND
80     aoRVLeadProximalTip.aoRVPort[back_of] /= covering AND
81     aoRVLeadProximalTip.aoRAPort[back_of] /= covering AND
82     aoRVSetScrew.mPulseGenerator[right_of] = touching AND
83     FORALL(x: directional): aoRALeadProximalTip.aoRVPort[x] = disjoint_to AND
84     FORALL(x: directional): aoLVLeadProximalTip.aoRVPort[x] = disjoint_to AND
```

```
85       pSurgeon_mPulseGenerator.orientable[pitch_back] = TRUE AND
86       pSurgeon_mPulseGenerator.orientable[pitch_forth] = TRUE AND
87       pSurgeon_mPulseGenerator.orientable[yaw_left] = TRUE AND
88       pSurgeon_mPulseGenerator.orientable[yaw_right] = TRUE AND
89       FORALL(x: translate): pSurgeon_mPulseGenerator.translatable[x] = TRUE AND
90       FORALL(x: position): pSurgeon_mPulseGenerator.positionable[x] = TRUE AND
91       pSurgeon_aoRVLeadProximalTip.orientable[pitch_back] = TRUE AND
92       pSurgeon_aoRVLeadProximalTip.orientable[pitch_forth] = TRUE AND
93       pSurgeon_aoRVLeadProximalTip.orientable[yaw_left] = TRUE AND
94       pSurgeon_aoRVLeadProximalTip.orientable[yaw_right] = TRUE AND
95       FORALL(x: translate): pSurgeon_aoRVLeadProximalTip.translatable[x] = TRUE AND
96       FORALL(x: position): pSurgeon_aoRVLeadProximalTip.positionable[x] = TRUE;
97
98     END;
99
100  HES: MODULE =
101      BEGIN
102       OUTPUT pSurgeon: pSurgeon_rels
103       OUTPUT aoLVLeadProximalTip: aoLVLeadProximalTip_rels
104       OUTPUT aoRVSetScrew: aoRVSetScrew_rels
105       OUTPUT aoLVSetScrew: aoLVSetScrew_rels
106       OUTPUT aoRALeadProximalTip: aoRALeadProximalTip_rels
107       OUTPUT aoRVLeadProximalTip: aoRVLeadProximalTip_rels
108       OUTPUT pSurgeon_mPulseGenerator: abilities
109       OUTPUT pSurgeon_aoLVLeadProximalTip: abilities
110       OUTPUT pSurgeon_aoRVLeadProximalTip: abilities
111       INPUT LVLeadConnectableToRVPort: BOOLEAN
112       INPUT LVLeadConnectableToLVPort: BOOLEAN
113       INPUT RVLeadConnectableToRVPort: BOOLEAN
114
115       INITIALIZATION
116         (FORALL(x: orient): pSurgeon_mPulseGenerator.orientable[x] = TRUE
117           AND pSurgeon_aoLVLeadProximalTip.orientable[x] = TRUE
118           AND pSurgeon_aoRVLeadProximalTip.orientable[x] = TRUE);
119         (FORALL(x: position): pSurgeon_mPulseGenerator.positionable[x] = TRUE
120           AND pSurgeon_aoLVLeadProximalTip.positionable[x] = TRUE
121           AND pSurgeon_aoRVLeadProximalTip.positionable[x] = TRUE);
122         (FORALL(x: translate): pSurgeon_mPulseGenerator.translatable[x] = TRUE
123           AND pSurgeon_aoLVLeadProximalTip.translatable[x] = TRUE
124           AND pSurgeon_aoRVLeadProximalTip.translatable[x] = TRUE);
125
126         pSurgeon IN {x: pSurgeon_rels |
127               (FORALL(y: directional, z: topological): x.aoRVPort[y] = z => (y /= front_of => z =
                      disjoint_to)) AND
128               (FORALL(y: directional, z: topological): x.aoLVPort[y] = z => (y /= front_of => z =
                      disjoint_to))};
129
130         aoLVLeadProximalTip IN {x: aoLVLeadProximalTip_rels |
131             (FORALL(q: directional, r: topological):
132               (q /= front_of AND r = overlapping) OR (q /= back_of AND r = covering)  =>
133                 not(x.aoRAPort[q] = r OR x.aoRVPort[q] = r OR x.aoLVPort[q] = r)) AND
134             not(EXISTS(y, z: topological, s, t: directional):
135               ((y = touching AND z = touching) OR (y = overlapping AND z = overlapping))
136                 AND ((s = top_of AND t = bottom_of) OR (s = left_of AND t = right_of) OR (s =
                        back_of OR t = back_of))
137                AND ((x.aoRVPort[s] = y AND x.aoRVPort[t] = z)
138                   OR (x.aoLVPort[s] = y AND x.aoLVPort[t] = z)
139                   OR (x.aoRAPort[s] = y AND x.aoRAPort[t] = z)))
140             AND ((EXISTS(y: directional): x.aoRVPort[y] /= disjoint_to) =>
141                   (FORALL(z: directional): x.aoLVPort[z] = disjoint_to AND x.aoRAPort[z] =
                          disjoint_to))
142             AND ((EXISTS(y: directional): x.aoLVPort[y] /= disjoint_to) =>
143                   (FORALL(z: directional): x.aoRVPort[z] = disjoint_to AND x.aoRAPort[z] =
                          disjoint_to))
144             AND ((EXISTS(y: directional): x.aoRAPort[y] /= disjoint_to) =>
145                   (FORALL(z: directional): x.aoLVPort[z] = disjoint_to AND x.aoRVPort[z] =
                          disjoint_to))};
146
147         aoRVLeadProximalTip IN {x: aoRVLeadProximalTip_rels |
148             (FORALL(q: directional, r: topological):
149               (q /= front_of AND r = overlapping) OR (q /= back_of AND r = covering)  =>
150                 not(x.aoRAPort[q] = r OR x.aoRVPort[q] = r OR x.aoLVPort[q] = r)) AND
151             not(EXISTS(y, z: topological, s, t: directional):
152               ((y = touching AND z = touching) OR (y = overlapping AND z = overlapping))
153                 AND ((s = top_of AND t = bottom_of) OR (s = left_of AND t = right_of) OR (s =
                        back_of OR t = back_of))
154                AND ((x.aoRVPort[s] = y AND x.aoRVPort[t] = z)
155                   OR (x.aoLVPort[s] = y AND x.aoLVPort[t] = z)
156                   OR (x.aoRAPort[s] = y AND x.aoRAPort[t] = z)))
157             AND ((EXISTS(y: directional): x.aoRVPort[y] /= disjoint_to) =>
158                   (FORALL(z: directional): x.aoLVPort[z] = disjoint_to AND x.aoRAPort[z] =
```

```
                     disjoint_to))
159          AND ((EXISTS(y: directional): x.aoLVPort[y] /= disjoint_to) =>
160                (FORALL(z: directional): x.aoRVPort[z] = disjoint_to AND x.aoRAPort[z] =
                     disjoint_to))
161          AND ((EXISTS(y: directional): x.aoRAPort[y] /= disjoint_to) =>
162                (FORALL(z: directional): x.aoLVPort[z] = disjoint_to AND x.aoRVPort[z] =
                     disjoint_to))};
163
164      aoRALeadProximalTip IN {x: aoRALeadProximalTip_rels |
165          (FORALL(q: directional, r: topological):
166            (q /= front_of AND r = overlapping) OR (q /= back_of AND r = covering) =>
167              not(x.aoRVPort[q] = r OR x.aoLVPort[q] = r)) AND
168          not(EXISTS(y, z: topological, s, t: directional):
169            ((y = touching AND z = touching) OR (y = overlapping AND z = overlapping)) AND
170              ((s = top_of AND t = bottom_of) OR (s = left_of AND t = right_of) OR (s = back_of
                   OR t = back_of))
171             AND ((x.aoRVPort[s] = y AND x.aoRVPort[t] = z)
172                OR (x.aoLVPort[s] = y AND x.aoLVPort[t] = z)))
173          AND ((EXISTS(y: directional): x.aoRVPort[y] /= disjoint_to) =>
174                (FORALL(z: directional): x.aoLVPort[z] = disjoint_to))
175          AND ((EXISTS(y: directional): x.aoLVPort[y] /= disjoint_to) =>
176                (FORALL(z: directional): x.aoRVPort[z] = disjoint_to))};
177
178      aoRVSetScrew IN {x: aoRVSetScrew_rels |
179        (FORALL(y: directional): (y = right_of => x.mPulseGenerator[y] = touching OR x.
             mPulseGenerator[y] = disjoint_to) AND
180                                (y /= right_of => x.mPulseGenerator[y] = disjoint_to))};
181      aoLVSetScrew IN {x: aoLVSetScrew_rels |
182        (FORALL(y: directional): (y = right_of => x.mPulseGenerator[y] = touching OR x.
             mPulseGenerator[y] = disjoint_to) AND
183                                (y /= right_of => x.mPulseGenerator[y] = disjoint_to))};
184
185        TRANSITION [
186            LVLeadConnectableToRVPort -->
187              (FORALL(d: directional):
188                aoLVLeadProximalTip'.aoRVPort[d] = covering;
189                aoLVLeadProximalTip'.aoLVPort[d] = disjoint_to;
190                aoLVLeadProximalTip'.aoRAPort[d] = disjoint_to);
191
192          []LVLeadConnectableToLVPort -->
193              (FORALL(d: directional):
194                aoLVLeadProximalTip'.aoLVPort[d] = covering;
195                aoLVLeadProximalTip'.aoRVPort[d] = disjoint_to;
196                aoLVLeadProximalTip'.aoRAPort[d] = disjoint_to);
197          []RVLeadConnectableToRVPort -->
198              (FORALL(d: directional):
199                aoRVLeadProximalTip'.aoRVPort[d] = covering;
200                aoRVLeadProximalTip'.aoLVPort[d] = disjoint_to;
201                aoRVLeadProximalTip'.aoRAPort[d] = disjoint_to);
202          []ELSE -->
203        ];
204  END;
205 affordances: MODULE = affordance || HES;
206 END
```

Appendix G: Chapter 7 Code Listing

G.1 XML Code

```
1  <bigsis xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2      xsi:noNamespaceSchemaLocation="../../schema/bigsis-2.0.xsd">
3      <signified-meanings name="PumpSpeed">Stopped, Low, Lowest, Medium, High, Highest,
           BelowFiveThousandRPM, AboveFiveThousandRPM, EightThousandRPM,
4          NineThousandRPM, TenThousandRPM, ElevenThousandRPM, TwelveThousandRPM</signified-meanings
               >
5      <signified-meanings name="PowerSupplied">ThreeToFourUnits, FourToFiveUnits, FiveToSixUnits,
           SixToSevenUnits,
6          SevenToEightUnits, EightToNineUnits, NineToTenUnits, TenToElevenUnits,
               ElevenToTwelveUnits,
7          ThreeToFourWatts, FourToFiveWatts, FiveToSixWatts, SixToSevenWatts, SevenToEightWatts,
8          EightToNineWatts, NineToTenWatts, TenToElevenWatts, ElevenToTwelveWatts,
9          FiveToSevenWatts, SevenToNineWatts, EightToTenWatts, ThirteenUnits,
               ThirteenWattsOrGreater</signified-meanings>
10     <signifier-properties of="PowerIndicators">
11         <Color signifies="PowerSupplied" when-colored="Green">PowerIndicators.Label.PowerSupplied
               </Color>
12         <Color signifies="PowerSupplied" when-colored="Amber">ThirteenUnits</Color>
13         <Label signifies="PowerSupplied" when-labeled="ThreeAndFour">ThreeToFourUnits</Label>
14         <Label signifies="PowerSupplied" when-labeled="FourAndFive">FourToFiveUnits</Label>
15         <Label signifies="PowerSupplied" when-labeled="FiveAndSix">FiveToSixUnits</Label>
16         <Label signifies="PowerSupplied" when-labeled="SixAndSeven">SixToSevenUnits</Label>
17         <Label signifies="PowerSupplied" when-labeled="SevenAndEight">SevenToEightUnits</Label>
18         <Label signifies="PowerSupplied" when-labeled="EightAndNine">EightToNineUnits</Label>
19         <Label signifies="PowerSupplied" when-labeled="NineAndTen">NineToTenUnits</Label>
20         <Label signifies="PowerSupplied" when-labeled="TenAndEleven">TenToElevenUnits</Label>
21         <Label signifies="PowerSupplied" when-labeled="ElevenAndTwelve">ElevenToTwelveUnits</
               Label>
22         <Label signifies="PowerSupplied" when-labeled="Thirteen">ThirteenUnits</Label>
23         <aPattern signifies="PowerSupplied" when-pattern="Continuous">ThirteenUnits</aPattern>
24         <Volume signifies="PowerSupplied" when-level="Loud">ThirteenUnits</Volume>
25     </signifier-properties>
26     <signifier-properties of="PumpStoppedAlarm">
27         <Color signifies="PumpSpeed" when-colored="Red">Stopped</Color>
28         <Color signifies="PumpSpeed" when-colored="noColor">SpeedSettingKnob.Label.PumpSpeed</
               Color>
29         <aPattern signifies="PumpSpeed" when-pattern="Continuous">Stopped</aPattern>
30         <Volume signifies="PumpSpeed" when-level="Loud">Stopped</Volume>
31     </signifier-properties>
32     <signifier-properties of="SpeedSettingKnob">
33         <Label signifies="PumpSpeed" when-labeled="One">Lowest</Label>
34         <Label signifies="PumpSpeed" when-labeled="Two">Low</Label>
35         <Label signifies="PumpSpeed" when-labeled="Three">Medium</Label>
36         <Label signifies="PumpSpeed" when-labeled="Four">High</Label>
37         <Label signifies="PumpSpeed" when-labeled="Five">Highest</Label>
38     </signifier-properties>
39     <property-documentation of="PowerIndicators">
40         <Color signifies="PumpSpeed" when-colored="Green">PowerIndicators.Label.PumpSpeed</Color>
41         <Color signifies="PowerSupplied" when-colored="Green">PowerIndicators.Label.PowerSupplied
               </Color>
42         <Color signifies="PowerSupplied" when-colored="Amber">ThirteenWattsOrGreater</Color>
43         <Label signifies="PowerSupplied" when-labeled="ThreeAndFour">ThreeToFourWatts</Label>
44         <Label signifies="PowerSupplied" when-labeled="FourAndFive">FourToFiveWatts</Label>
45         <Label signifies="PowerSupplied" when-labeled="FiveAndSix">FiveToSixWatts</Label>
46         <Label signifies="PowerSupplied" when-labeled="SixAndSeven">SixToSevenWatts</Label>
47         <Label signifies="PowerSupplied" when-labeled="SevenAndEight">SevenToEightWatts</Label>
48         <Label signifies="PowerSupplied" when-labeled="EightAndNine">EightToNineWatts</Label>
49         <Label signifies="PowerSupplied" when-labeled="NineAndTen">NineToTenWatts</Label>
50         <Label signifies="PowerSupplied" when-labeled="TenAndEleven">TenToElevenWatts</Label>
51         <Label signifies="PowerSupplied" when-labeled="ElevenAndTwelve">ElevenToTwelveWatts</
               Label>
52         <Label signifies="PowerSupplied" when-labeled="Thirteen">ThirteenWattsOrGreater</Label>
53         <Label signifies="PumpSpeed" when-labeled="ThreeAndFour">EightThousandRPM</Label>
54         <Label signifies="PumpSpeed" when-labeled="FourAndFive">NineThousandRPM</Label>
55         <Label signifies="PumpSpeed" when-labeled="FiveAndSix">TenThousandRPM</Label>
56         <Label signifies="PumpSpeed" when-labeled="SixAndSeven">TenThousandRPM</Label>
57         <Label signifies="PumpSpeed" when-labeled="SevenAndEight">ElevenThousandRPM</Label>
58         <Label signifies="PumpSpeed" when-labeled="NineAndTen">TwelveThousandRPM</Label>
```

```
59          <aPattern signifies="PowerSupplied" when-pattern="Continuous">ThirteenWattsOrGreater</
                aPattern>
60          <Volume signifies="PowerSupplied" when-level="Loud">ThirteenWattsOrGreater</Volume>
61      </property-documentation>
62      <property-documentation of="PumpStoppedAlarm">
63          <Color signifies="PumpSpeed" when-colored="Red">BelowFiveThousandRPM</Color>
64          <aPattern signifies="PumpSpeed" when-pattern="Continuous">BelowFiveThousandRPM</aPattern>
65          <Volume signifies="PumpSpeed" when-level="Loud">BelowFiveThousandRPM</Volume>
66      </property-documentation>
67      <property-documentation of="SpeedSettingKnob">
68          <Label signifies="PumpSpeed" when-labeled="One">EightThousandRPM</Label>
69          <Label signifies="PumpSpeed" when-labeled="Two">NineThousandRPM</Label>
70          <Label signifies="PumpSpeed" when-labeled="Three">TenThousandRPM</Label>
71          <Label signifies="PumpSpeed" when-labeled="Four">ElevenThousandRPM</Label>
72          <Label signifies="PumpSpeed" when-labeled="Five">TwelveThousandRPM</Label>
73          <Label signifies="PowerSupplied" when-labeled="One">ThreeToFourWatts</Label>
74          <Label signifies="PowerSupplied" when-labeled="Two">FourToFiveWatts</Label>
75          <Label signifies="PowerSupplied" when-labeled="Three">FiveToSevenWatts</Label>
76          <Label signifies="PowerSupplied" when-labeled="Four">SevenToNineWatts</Label>
77          <Label signifies="PowerSupplied" when-labeled="Five">EightToTenWatts</Label>
78      </property-documentation>
79  </bigsis>
```

G.2 SAL Code

G.2.1 BIGSIS-SAL Model

```
1   bigsis: CONTEXT =
2   BEGIN
3     PumpSpeed: TYPE = {PumpSpeedNotSignified, Stopped, Low, Lowest, Medium, High, Highest,
          BelowFiveThousandRPM, AboveFiveThousandRPM, EightThousandRPM,
4           NineThousandRPM, TenThousandRPM, ElevenThousandRPM, TwelveThousandRPM};
5     PowerSupplied: TYPE = {PowerSuppliedNotSignified, ThreeToFourUnits, FourToFiveUnits,
          FiveToSixUnits, SixToSevenUnits,
6           SevenToEightUnits, EightToNineUnits, NineToTenUnits, TenToElevenUnits,
                ElevenToTwelveUnits,
7           ThreeToFourWatts, FourToFiveWatts, FiveToSixWatts, SixToSevenWatts, SevenToEightWatts,
8           EightToNineWatts, NineToTenWatts, TenToElevenWatts, ElevenToTwelveWatts,
9           FiveToSevenWatts, SevenToNineWatts, EightToTenWatts, ThirteenUnits,
                ThirteenWattsOrGreater};
10    COLORS: TYPE = {Green,Amber,Red,noColor};
11    colorSwitch(x: [BOOLEAN->COLORS]): COLORS;
12    LABELS: TYPE = {ThreeAndFour,FourAndFive,FiveAndSix,SixAndSeven,SevenAndEight,EightAndNine,
          NineAndTen,TenAndEleven,ElevenAndTwelve,Thirteen,One,Two,Three,Four,Five,noLabel};
13    PTRN: TYPE = {Continuous,noPattern};
14    VOL: TYPE = {Loud,noVolume};
15    ORNT: TYPE = {noOrientation, up, down, left, right, away, toward};
16    Colors_signify: TYPE = [#Colored: COLORS, PowerSupplied: PowerSupplied, PumpSpeed: PumpSpeed#];
17    Labels_signify: TYPE = [#Labeled: LABELS, PowerSupplied: PowerSupplied, PumpSpeed: PumpSpeed#];
18    aPatterns_signify: TYPE = [#Pattern: PTRN, PowerSupplied: PowerSupplied, PumpSpeed: PumpSpeed
          #];
19    Volumes_signify: TYPE = [#Level: VOL, PowerSupplied: PowerSupplied, PumpSpeed: PumpSpeed#];
20    PowerIndicators_signifiers: TYPE = [#Color: Colors_signify, Label: Labels_signify, aPattern:
          aPatterns_signify, Volume: Volumes_signify#];
21    PumpStoppedAlarm_signifiers: TYPE = [#Color: Colors_signify, aPattern: aPatterns_signify,
          Volume: Volumes_signify#];
22    SpeedSettingKnob_signifiers: TYPE = [#Label: Labels_signify#];
23    Doc_PowerIndicators_signifiers: TYPE = [#Color: Colors_signify, Label: Labels_signify, aPattern
          : aPatterns_signify, Volume: Volumes_signify#];
24    Doc_PumpStoppedAlarm_signifiers: TYPE = [#Color: Colors_signify, aPattern: aPatterns_signify,
          Volume: Volumes_signify#];
25    Doc_SpeedSettingKnob_signifiers: TYPE = [#Label: Labels_signify#];
26  signifiers: MODULE =
27   BEGIN
28     LOCAL PowerIndicators: PowerIndicators_signifiers
29     LOCAL PumpStoppedAlarm: PumpStoppedAlarm_signifiers
30     LOCAL SpeedSettingKnob: SpeedSettingKnob_signifiers
31     LOCAL Doc_PowerIndicators: Doc_PowerIndicators_signifiers
32     LOCAL Doc_PumpStoppedAlarm: Doc_PumpStoppedAlarm_signifiers
33     LOCAL Doc_SpeedSettingKnob: Doc_SpeedSettingKnob_signifiers
34     INPUT action: bigsisDeviceModel!rotations
35     INPUT alarm: bigsisDeviceModel!alarms
36     LOCAL PowerLabel: LABELS
37       DEFINITION
38         PowerLabel IN {ThreeAndFour,FourAndFive,FiveAndSix,SixAndSeven,SevenAndEight,EightAndNine,
               NineAndTen,TenAndEleven,ElevenAndTwelve};
39     INITIALIZATION
40       PowerIndicators.Color.Colored = Green;
41       PowerIndicators.Volume.Level = Loud;
```

```
42        PowerIndicators.aPattern.Pattern = Continuous;
43        PumpStoppedAlarm.Color.Colored = Red;
44        PumpStoppedAlarm.Volume.Level = Loud;
45        PumpStoppedAlarm.aPattern.Pattern = Continuous;
46        SpeedSettingKnob.Label.Labeled = Four;
47        PowerIndicators.Label.Labeled = EightAndNine;
48        TRANSITION
49          [
50          alarm' = bigsisDeviceModel!HighPower -->
51             PowerIndicators'.Color.Colored = Amber;
52             PowerIndicators'.Label.Labeled = Thirteen;
53             PowerIndicators'.Volume.Level = Loud;
54             PowerIndicators'.aPattern.Pattern = Continuous;
55             PumpStoppedAlarm'.Color.Colored = noColor;
56             PumpStoppedAlarm'.Volume.Level = Loud;
57             PumpStoppedAlarm'.aPattern.Pattern = Continuous;
58          []alarm' = bigsisDeviceModel!PumpStopped -->
59             PowerIndicators'.Color.Colored = Green;
60             PowerIndicators'.Label.Labeled = PowerLabel;
61             PowerIndicators'.Volume.Level = Loud;
62             PowerIndicators'.aPattern.Pattern = Continuous;
63             PumpStoppedAlarm'.Color.Colored = Red;
64             PumpStoppedAlarm'.Volume.Level = Loud;
65             PumpStoppedAlarm'.aPattern.Pattern = Continuous;
66          []alarm' = bigsisDeviceModel!NoAlarm -->
67             PowerIndicators'.Color.Colored = Green;
68             PowerIndicators'.Label.Labeled = PowerLabel;
69             PowerIndicators'.Volume.Level = noVolume;
70             PowerIndicators'.aPattern.Pattern = noPattern;
71             PumpStoppedAlarm'.Color.Colored = noColor;
72             PumpStoppedAlarm'.Volume.Level = noVolume;
73             PumpStoppedAlarm'.aPattern.Pattern = noPattern;
74          []action' = bigsisDeviceModel!increaseSpeed -->
75              SpeedSettingKnob'.Label.Labeled = IF SpeedSettingKnob.Label.Labeled = One THEN Two
76               ELSIF SpeedSettingKnob.Label.Labeled = Two THEN Three
77               ELSIF SpeedSettingKnob.Label.Labeled = Three THEN Four
78               ELSE Five
79             ENDIF;
80           []action' = bigsisDeviceModel!decreaseSpeed -->
81               SpeedSettingKnob'.Label.Labeled = IF SpeedSettingKnob.Label.Labeled = Two THEN One
82                ELSIF SpeedSettingKnob.Label.Labeled = Three THEN Two
83                ELSIF SpeedSettingKnob.Label.Labeled = Four THEN Three
84                ELSE Four
85              ENDIF;
86          []ELSE -->
87        ];
88
89     INITIALIZATION
90       PowerIndicators.Color.PowerSupplied = IF PowerIndicators.Color.Colored = Green THEN
             PowerIndicators.Label.PowerSupplied ELSIF PowerIndicators.Color.Colored = Amber THEN
             ThirteenUnits ELSE PowerSuppliedNotSignified ENDIF;
91       PowerIndicators.Label.PowerSupplied = IF PowerIndicators.Label.Labeled = ThreeAndFour THEN
             ThreeToFourUnits ELSIF PowerIndicators.Label.Labeled = FourAndFive THEN FourToFiveUnits
             ELSIF PowerIndicators.Label.Labeled = FiveAndSix THEN FiveToSixUnits ELSIF
             PowerIndicators.Label.Labeled = SixAndSeven THEN SixToSevenUnits ELSIF PowerIndicators.
             Label.Labeled = SevenAndEight THEN SevenToEightUnits ELSIF PowerIndicators.Label.Labeled
              = EightAndNine THEN EightToNineUnits ELSIF PowerIndicators.Label.Labeled = NineAndTen
             THEN NineToTenUnits ELSIF PowerIndicators.Label.Labeled = TenAndEleven THEN
             TenToElevenUnits ELSIF PowerIndicators.Label.Labeled = ElevenAndTwelve THEN
             ElevenToTwelveUnits ELSIF PowerIndicators.Label.Labeled = Thirteen THEN ThirteenUnits
             ELSE PowerSuppliedNotSignified ENDIF;
92       PowerIndicators.aPattern.PowerSupplied = IF PowerIndicators.aPattern.Pattern = Continuous
             THEN ThirteenUnits ELSE PowerSuppliedNotSignified ENDIF;
93       PowerIndicators.Volume.PowerSupplied = IF PowerIndicators.Volume.Level = Loud THEN
             ThirteenUnits ELSE PowerSuppliedNotSignified ENDIF;
94       PumpStoppedAlarm.Color.PumpSpeed = IF PumpStoppedAlarm.Color.Colored = Red THEN Stopped ELSIF
              PumpStoppedAlarm.Color.Colored = noColor THEN SpeedSettingKnob.Label.PumpSpeed ELSE
             PumpSpeedNotSignified ENDIF;
95       PumpStoppedAlarm.aPattern.PumpSpeed = IF PumpStoppedAlarm.aPattern.Pattern = Continuous THEN
             Stopped ELSE PumpSpeedNotSignified ENDIF;
96       PumpStoppedAlarm.Volume.PumpSpeed = IF PumpStoppedAlarm.Volume.Level = Loud THEN Stopped ELSE
              PumpSpeedNotSignified ENDIF;
97       SpeedSettingKnob.Label.PumpSpeed = IF SpeedSettingKnob.Label.Labeled = One THEN Lowest ELSIF
             SpeedSettingKnob.Label.Labeled = Two THEN Low ELSIF SpeedSettingKnob.Label.Labeled =
             Three THEN Medium ELSIF SpeedSettingKnob.Label.Labeled = Four THEN High ELSIF
             SpeedSettingKnob.Label.Labeled = Five THEN Highest ELSE PumpSpeedNotSignified ENDIF;
98       Doc_PowerIndicators.Color.PumpSpeed = IF PowerIndicators.Color.Colored = Green THEN
             Doc_PowerIndicators.Label.PumpSpeed ELSE PumpSpeedNotSignified ENDIF;
99       Doc_PowerIndicators.Color.PowerSupplied = IF PowerIndicators.Color.Colored = Green THEN
             Doc_PowerIndicators.Label.PowerSupplied ELSIF PowerIndicators.Color.Colored = Amber THEN
              ThirteenWattsOrGreater ELSE PowerSuppliedNotSignified ENDIF;
```

```
100      Doc_PowerIndicators.Label.PowerSupplied = IF PowerIndicators.Label.Labeled = ThreeAndFour
             THEN ThreeToFourWatts ELSIF PowerIndicators.Label.Labeled = FourAndFive THEN
             FourToFiveWatts ELSIF PowerIndicators.Label.Labeled = FiveAndSix THEN FiveToSixWatts
             ELSIF PowerIndicators.Label.Labeled = SixAndSeven THEN SixToSevenWatts ELSIF
             PowerIndicators.Label.Labeled = SevenAndEight THEN SevenToEightWatts ELSIF
             PowerIndicators.Label.Labeled = EightAndNine THEN EightToNineWatts ELSIF PowerIndicators
             .Label.Labeled = NineAndTen THEN NineToTenWatts ELSIF PowerIndicators.Label.Labeled =
             TenAndEleven THEN TenToElevenWatts ELSIF PowerIndicators.Label.Labeled = ElevenAndTwelve
              THEN ElevenToTwelveWatts ELSIF PowerIndicators.Label.Labeled = Thirteen THEN
             ThirteenWattsOrGreater ELSE PowerSuppliedNotSignified ENDIF;
101      Doc_PowerIndicators.Label.PumpSpeed = IF PowerIndicators.Label.Labeled = ThreeAndFour THEN
             EightThousandRPM ELSIF PowerIndicators.Label.Labeled = FourAndFive THEN NineThousandRPM
             ELSIF PowerIndicators.Label.Labeled = FiveAndSix THEN TenThousandRPM ELSIF
             PowerIndicators.Label.Labeled = SixAndSeven THEN TenThousandRPM ELSIF PowerIndicators.
             Label.Labeled = SevenAndEight THEN ElevenThousandRPM ELSIF PowerIndicators.Label.Labeled
              = NineAndTen THEN TwelveThousandRPM ELSE PumpSpeedNotSignified ENDIF;
102      Doc_PowerIndicators.aPattern.PowerSupplied = IF PowerIndicators.aPattern.Pattern = Continuous
              THEN ThirteenWattsOrGreater ELSE PowerSuppliedNotSignified ENDIF;
103      Doc_PowerIndicators.Volume.PowerSupplied = IF PowerIndicators.Volume.Level = Loud THEN
             ThirteenWattsOrGreater ELSE PowerSuppliedNotSignified ENDIF;
104      Doc_PumpStoppedAlarm.Color.PumpSpeed = IF PumpStoppedAlarm.Color.Colored = Red THEN
             BelowFiveThousandRPM ELSE PumpSpeedNotSignified ENDIF;
105      Doc_PumpStoppedAlarm.aPattern.PumpSpeed = IF PumpStoppedAlarm.aPattern.Pattern = Continuous
             THEN BelowFiveThousandRPM ELSE PumpSpeedNotSignified ENDIF;
106      Doc_PumpStoppedAlarm.Volume.PumpSpeed = IF PumpStoppedAlarm.Volume.Level = Loud THEN
             BelowFiveThousandRPM ELSE PumpSpeedNotSignified ENDIF;
107      Doc_SpeedSettingKnob.Label.PumpSpeed = IF SpeedSettingKnob.Label.Labeled = One THEN
             EightThousandRPM ELSIF SpeedSettingKnob.Label.Labeled = Two THEN NineThousandRPM ELSIF
             SpeedSettingKnob.Label.Labeled = Three THEN TenThousandRPM ELSIF SpeedSettingKnob.Label.
             Labeled = Four THEN ElevenThousandRPM ELSIF SpeedSettingKnob.Label.Labeled = Five THEN
             TwelveThousandRPM ELSE PumpSpeedNotSignified ENDIF;
108      Doc_SpeedSettingKnob.Label.PowerSupplied = IF SpeedSettingKnob.Label.Labeled = One THEN
             ThreeToFourWatts ELSIF SpeedSettingKnob.Label.Labeled = Two THEN FourToFiveWatts ELSIF
             SpeedSettingKnob.Label.Labeled = Three THEN FiveToSevenWatts ELSIF SpeedSettingKnob.
             Label.Labeled = Four THEN SevenToNineWatts ELSIF SpeedSettingKnob.Label.Labeled = Five
             THEN EightToTenWatts ELSE PowerSuppliedNotSignified ENDIF;
109
110    TRANSITION
111      PowerIndicators'.Color.PowerSupplied = IF PowerIndicators'.Color.Colored = Green THEN
             PowerIndicators'.Label.PowerSupplied ELSIF PowerIndicators'.Color.Colored = Amber THEN
             ThirteenUnits ELSE PowerSuppliedNotSignified ENDIF;
112      PowerIndicators'.Label.PowerSupplied = IF PowerIndicators'.Label.Labeled = ThreeAndFour THEN
             ThreeToFourUnits ELSIF PowerIndicators'.Label.Labeled = FourAndFive THEN FourToFiveUnits
              ELSIF PowerIndicators'.Label.Labeled = FiveAndSix THEN FiveToSixUnits ELSIF
             PowerIndicators'.Label.Labeled = SixAndSeven THEN SixToSevenUnits ELSIF PowerIndicators
             '.Label.Labeled = SevenAndEight THEN SevenToEightUnits ELSIF PowerIndicators'.Label.
             Labeled = EightAndNine THEN EightToNineUnits ELSIF PowerIndicators'.Label.Labeled =
             NineAndTen THEN NineToTenUnits ELSIF PowerIndicators'.Label.Labeled = TenAndEleven THEN
             TenToElevenUnits ELSIF PowerIndicators'.Label.Labeled = ElevenAndTwelve THEN
             ElevenToTwelveUnits ELSIF PowerIndicators'.Label.Labeled = Thirteen THEN ThirteenUnits
             ELSE PowerSuppliedNotSignified ENDIF;
113      PowerIndicators'.aPattern.PowerSupplied = IF PowerIndicators'.aPattern.Pattern = Continuous
             THEN ThirteenUnits ELSE PowerSuppliedNotSignified ENDIF;
114      PowerIndicators'.Volume.PowerSupplied = IF PowerIndicators'.Volume.Level = Loud THEN
             ThirteenUnits ELSE PowerSuppliedNotSignified ENDIF;
115      PumpStoppedAlarm'.Color.PumpSpeed = IF PumpStoppedAlarm'.Color.Colored = Red THEN Stopped
             ELSIF PumpStoppedAlarm'.Color.Colored = noColor THEN SpeedSettingKnob'.Label.PumpSpeed
             ELSE PumpSpeedNotSignified ENDIF;
116      PumpStoppedAlarm'.aPattern.PumpSpeed = IF PumpStoppedAlarm'.aPattern.Pattern = Continuous
             THEN Stopped ELSE PumpSpeedNotSignified ENDIF;
117      PumpStoppedAlarm'.Volume.PumpSpeed = IF PumpStoppedAlarm'.Volume.Level = Loud THEN Stopped
             ELSE PumpSpeedNotSignified ENDIF;
118      SpeedSettingKnob'.Label.PumpSpeed = IF SpeedSettingKnob'.Label.Labeled = One THEN Lowest
             ELSIF SpeedSettingKnob'.Label.Labeled = Two THEN Low ELSIF SpeedSettingKnob'.Label.
             Labeled = Three THEN Medium ELSIF SpeedSettingKnob'.Label.Labeled = Four THEN High ELSIF
              SpeedSettingKnob'.Label.Labeled = Five THEN Highest ELSE PumpSpeedNotSignified ENDIF;
119      Doc_PowerIndicators'.Color.PumpSpeed = IF PowerIndicators'.Color.Colored = Green THEN
             Doc_PowerIndicators'.Label.PumpSpeed ELSE PumpSpeedNotSignified ENDIF;
120      Doc_PowerIndicators'.Color.PowerSupplied = IF PowerIndicators'.Color.Colored = Green THEN
             Doc_PowerIndicators'.Label.PowerSupplied ELSIF PowerIndicators'.Color.Colored = Amber
             THEN ThirteenWattsOrGreater ELSE PowerSuppliedNotSignified ENDIF;
121      Doc_PowerIndicators'.Label.PowerSupplied = IF PowerIndicators'.Label.Labeled = ThreeAndFour
             THEN ThreeToFourWatts ELSIF PowerIndicators'.Label.Labeled = FourAndFive THEN
             FourToFiveWatts ELSIF PowerIndicators'.Label.Labeled = FiveAndSix THEN FiveToSixWatts
             ELSIF PowerIndicators'.Label.Labeled = SixAndSeven THEN SixToSevenWatts ELSIF
             PowerIndicators'.Label.Labeled = SevenAndEight THEN SevenToEightWatts ELSIF
             PowerIndicators'.Label.Labeled = EightAndNine THEN EightToNineWatts ELSIF
             PowerIndicators'.Label.Labeled = NineAndTen THEN NineToTenWatts ELSIF PowerIndicators'.
             Label.Labeled = TenAndEleven THEN TenToElevenWatts ELSIF PowerIndicators'.Label.Labeled
             = ElevenAndTwelve THEN ElevenToTwelveWatts ELSIF PowerIndicators'.Label.Labeled =
             Thirteen THEN ThirteenWattsOrGreater ELSE PowerSuppliedNotSignified ENDIF;
```

```
122      Doc_PowerIndicators'.Label.PumpSpeed = IF PowerIndicators'.Label.Labeled = ThreeAndFour THEN
             EightThousandRPM ELSIF PowerIndicators'.Label.Labeled = FourAndFive THEN NineThousandRPM
              ELSIF PowerIndicators'.Label.Labeled = FiveAndSix THEN TenThousandRPM ELSIF
             PowerIndicators'.Label.Labeled = SixAndSeven THEN TenThousandRPM ELSIF PowerIndicators'.
             Label.Labeled = SevenAndEight THEN ElevenThousandRPM ELSIF PowerIndicators'.Label.
             Labeled = NineAndTen THEN TwelveThousandRPM ELSE PumpSpeedNotSignified ENDIF;
123      Doc_PowerIndicators'.aPattern.PowerSupplied = IF PowerIndicators'.aPattern.Pattern =
             Continuous THEN ThirteenWattsOrGreater ELSE PowerSuppliedNotSignified ENDIF;
124      Doc_PowerIndicators'.Volume.PowerSupplied = IF PowerIndicators'.Volume.Level = Loud THEN
             ThirteenWattsOrGreater ELSE PowerSuppliedNotSignified ENDIF;
125      Doc_PumpStoppedAlarm'.Color.PumpSpeed = IF PumpStoppedAlarm'.Color.Colored = Red THEN
             BelowFiveThousandRPM ELSE PumpSpeedNotSignified ENDIF;
126      Doc_PumpStoppedAlarm'.aPattern.PumpSpeed = IF PumpStoppedAlarm'.aPattern.Pattern = Continuous
              THEN BelowFiveThousandRPM ELSE PumpSpeedNotSignified ENDIF;
127      Doc_PumpStoppedAlarm'.Volume.PumpSpeed = IF PumpStoppedAlarm'.Volume.Level = Loud THEN
             BelowFiveThousandRPM ELSE PumpSpeedNotSignified ENDIF;
128      Doc_SpeedSettingKnob'.Label.PumpSpeed = IF SpeedSettingKnob'.Label.Labeled = One THEN
             EightThousandRPM ELSIF SpeedSettingKnob'.Label.Labeled = Two THEN NineThousandRPM ELSIF
             SpeedSettingKnob'.Label.Labeled = Three THEN TenThousandRPM ELSIF SpeedSettingKnob'.
             Label.Labeled = Four THEN ElevenThousandRPM ELSIF SpeedSettingKnob'.Label.Labeled = Five
              THEN TwelveThousandRPM ELSE PumpSpeedNotSignified ENDIF;
129      Doc_SpeedSettingKnob'.Label.PowerSupplied = IF SpeedSettingKnob'.Label.Labeled = One THEN
             ThreeToFourWatts ELSIF SpeedSettingKnob'.Label.Labeled = Two THEN FourToFiveWatts ELSIF
             SpeedSettingKnob'.Label.Labeled = Three THEN FiveToSevenWatts ELSIF SpeedSettingKnob'.
             Label.Labeled = Four THEN SevenToNineWatts ELSIF SpeedSettingKnob'.Label.Labeled = Five
             THEN EightToTenWatts ELSE PowerSuppliedNotSignified ENDIF;
130
131     OUTPUT Visually_Signified_PumpSpeed: PumpSpeed
132     OUTPUT Audibly_Signified_PumpSpeed: PumpSpeed
133     OUTPUT Documented_PumpSpeed: PumpSpeed
134     OUTPUT Visually_Signified_PowerSupplied: PowerSupplied
135     OUTPUT Audibly_Signified_PowerSupplied: PowerSupplied
136     OUTPUT Documented_PowerSupplied: PowerSupplied
137
138   DEFINITION
139       Visually_Signified_PumpSpeed IN {PumpStoppedAlarm.Color.PumpSpeed, SpeedSettingKnob.Label.
              PumpSpeed};
140       Visually_Signified_PowerSupplied IN {PowerIndicators.Color.PowerSupplied, PowerIndicators.
              Label.PowerSupplied};
141       Audibly_Signified_PumpSpeed IN {PumpStoppedAlarm.aPattern.PumpSpeed, PumpStoppedAlarm.Volume
              .PumpSpeed};
142       Audibly_Signified_PowerSupplied IN {PowerIndicators.aPattern.PowerSupplied, PowerIndicators.
              Volume.PowerSupplied};
143       Documented_PumpSpeed IN {Doc_PowerIndicators.Color.PumpSpeed, Doc_PowerIndicators.Label.
              PumpSpeed, Doc_PumpStoppedAlarm.Color.PumpSpeed, Doc_PumpStoppedAlarm.aPattern.
              PumpSpeed, Doc_PumpStoppedAlarm.Volume.PumpSpeed, Doc_SpeedSettingKnob.Label.PumpSpeed
              };
144       Documented_PowerSupplied IN {Doc_PowerIndicators.Color.PowerSupplied, Doc_PowerIndicators.
              Label.PowerSupplied, Doc_PowerIndicators.aPattern.PowerSupplied, Doc_PowerIndicators.
              Volume.PowerSupplied, Doc_SpeedSettingKnob.Label.PowerSupplied};
145   END;
146
147 END
```

G.2.2 Device Model

```
 1 bigsisDeviceModel: CONTEXT =
 2   BEGIN
 3   alarms : TYPE = {NoAlarm, UnderSpeed, PumpStopped, HighPower};
 4   rotations: TYPE = {increaseSpeed, decreaseSpeed, none};
 5
 6   device: MODULE =
 7   BEGIN
 8     OUTPUT action: rotations
 9     OUTPUT alarm: alarms
10
11     INITIALIZATION
12       alarm = PumpStopped;
13       action = none;
14
15     TRANSITION
16       alarm' IN
17         IF alarm = NoAlarm
18           THEN {PumpStopped, HighPower, NoAlarm}
19         ELSIF alarm = PumpStopped
20           THEN {NoAlarm, HighPower}
21         ELSE {NoAlarm, PumpStopped}
22         ENDIF;
23       [
24         action = none -->
```

```
25             action' IN {increaseSpeed, decreaseSpeed};
26         []action = increaseSpeed -->
27             action' IN {decreaseSpeed, none};
28         []action = decreaseSpeed -->
29             action' IN {increaseSpeed, none};
30         []ELSE -->
31       ];
32   END;
33 END
```

Appendix H: Chapter 9 Code Listing

H.1 XML Code

H.1.1 Task Models

In the EOFM-XML code shown below, variable declarations are on lines 4–60. The EOFM task model representing the pump speed adjustment procedure is on lines 61–82. The EOFM task model representing the pump stopped alarm troubleshooting procedure is on lines 83–240.

```
<?xml version="1.0" encoding="UTF-8"?>
<eofms xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="../../../../Research/git/schema/OFMr8.xsd">
    <userdefinedtype name="tSpeed">[BOOLEAN -> INTEGER]</userdefinedtype>
    <userdefinedtype name="tRotationCounter">[0..3]</userdefinedtype>
    <userdefinedtype name="tOldComponentsLocation">{AtHand, SetAside}</userdefinedtype>
    <userdefinedtype name="tConnection">{Connected, Disconnected}</userdefinedtype>
    <userdefinedtype name="tPartTag">{redTagged, notRedTagged}</userdefinedtype>
    <userdefinedtype name="tPermAttachedConnectorStatus">{Broken, Assembled}</userdefinedtype>
    <userdefinedtype name="tAlarmBatteryCap">{Loosened, Tightened}</userdefinedtype>
    <humanoperator name="pPumpOperator">
        <inputvariable name="iPage" basictype="{x: INTEGER | x &gt;= 0 AND x &lt;= 29}"/>
        <inputvariable name="iAlarm" basictype="discreteDevice!Alarms"/>
        <inputvariable name="iSpeedSetting" basictype="discreteDevice!SpeedSettings"/>
        <inputvariable name="iPowerLight" basictype="discreteDevice!PowerLights"/>
        <inputvariable name="iSpeedSettingKnobLight" basictype="discreteDevice!KnobLight"/>
        <inputvariable name="iOldComponentTags" userdefinedtype="tPartTag"/>
        <inputvariable name="iLeadBattToOldController" userdefinedtype="tConnection"/>
        <inputvariable name="iLeadBattToNewController" userdefinedtype="tConnection"/>
        <inputvariable name="iLeadBattToYCable" userdefinedtype="tConnection"/>
        <inputvariable name="iYCableToOldController" userdefinedtype="tConnection"/>
        <inputvariable name="iOldLiBattCableToYCable" userdefinedtype="tConnection"/>
        <inputvariable name="iOldLiBattCableToOldController" userdefinedtype="tConnection"/>
        <inputvariable name="iOldLiBattCableToOldLiBatt" userdefinedtype="tConnection"/>
        <inputvariable name="iPumpCableToOldController" userdefinedtype="tConnection"/>
        <inputvariable name="iAbCableToOldController" userdefinedtype="tConnection"/>
        <inputvariable name="iNewLiBattCableToNewLiBatt" userdefinedtype="tConnection"/>
        <inputvariable name="iNewLiBattCableToNewController" userdefinedtype="tConnection"/>
        <inputvariable name="iPumpCableToOldAbCable" userdefinedtype="tConnection"/>
        <inputvariable name="iPumpCableToNewController" userdefinedtype="tConnection"/>
        <inputvariable name="iNewLiBatteryLights" basictype="discreteDevice!BatteryLights"/>
        <inputvariable name="iPermanentlyAttachedConnector" userdefinedtype="
            tPermAttachedConnectorStatus"/>
        <inputvariable name="iRotationCounter" userdefinedtype="tRotationCounter"/>
        <inputvariable name="iNewControllerAlarmBatteryCap" userdefinedtype="tAlarmBatteryCap"/>
        <inputvariable name="iOldControllerAlarmBatteryCap" userdefinedtype="tAlarmBatteryCap"/>
        <inputvariable name="iOldComponents" userdefinedtype="tOldComponentsLocation"/>
        <localvariable name="lDesiredSpeed" basictype="discreteDevice!SpeedSettings">
            <initialvalue>IN {1, 2, 3, 4, 5}</initialvalue>
        </localvariable>
        <humanaction name="hCallEmergencyNumber" behavior="autoreset"/>
        <humanaction name="hRedTagOldComponents" behavior="autoreset"/>
        <humanaction name="hSetAsideOldComponents" behavior="autoreset"/>
        <humanaction name="hRotateConnectorParts" behavior="autoreset"/>
        <humanaction name="hDisassembleConnector" behavior="autoreset"/>
        <humanaction name="hReassembleBrokenConnector" behavior="autoreset"/>
        <humanaction name="hDiscPumpCableFromAbCable" behavior="autoreset"/>
        <humanaction name="hDiscPumpCableFromOldController" behavior="autoreset"/>
        <humanaction name="hDiscLeadBattFromNewController" behavior="autoreset"/>
        <humanaction name="hDiscNewLiBattCableFromNewController" behavior="autoreset"/>
        <humanaction name="hConNewLiBattCableToNewController" behavior="autoreset"/>
        <humanaction name="hConNewLiBattCableToNewLiBatt" behavior="autoreset"/>
        <humanaction name="hConLeadBattToNewController" behavior="autoreset"/>
        <humanaction name="hConPumpCableToNewController" behavior="autoreset"/>
        <humanaction name="hDepressBlackButtonOnNewLiBatt" behavior="autoreset"/>
        <humanaction name="hRotateKnobClockwise" behavior="autoreset"/>
        <humanaction name="hRotateKnobCounterclockwise" behavior="autoreset"/>
        <humanaction name="hTightenNewControllerABCap" behavior="autoreset"/>
        <humanaction name="hLoosenOldControllerABCap" behavior="autoreset"/>
        <humanaction name="hLoosenNewControllerABCap" behavior="autoreset"/>
        <eofm>
            <activity name="aAdjustSpeed">
```

```
62                <precondition>iSpeedSetting /= lDesiredSpeed AND iAlarm = discreteDevice!NoAlarm<
                      /precondition>
63                <completioncondition>iSpeedSetting = lDesiredSpeed</completioncondition>
64                <decomposition operator="xor">
65                    <activity name="aIncreaseSetting">
66                        <precondition>iSpeedSetting &lt; lDesiredSpeed AND iPage = 8</
                              precondition>
67                        <repeatcondition>iSpeedSetting &lt; lDesiredSpeed</repeatcondition>
68                        <decomposition operator="ord">
69                            <action humanaction="hRotateKnobCounterclockwise"/>
70                        </decomposition>
71                    </activity>
72                    <activity name="aDecreaseSetting">
73                        <precondition>iSpeedSetting &gt; lDesiredSpeed AND iPage = 8</
                              precondition>
74                        <repeatcondition>iSpeedSetting &gt; lDesiredSpeed</repeatcondition>
75                        <decomposition operator="ord">
76                            <action humanaction="hRotateKnobClockwise"/>
77                        </decomposition>
78                    </activity>
79                </decomposition>
80            </activity>
81        </eofm>
82        <eofm>
83            <activity name="aRespondToPumpStoppedAlarm">
84                <precondition>iAlarm = discreteDevice!PumpStopped</precondition>
85                <completioncondition>iAlarm /= discreteDevice!PumpStopped</completioncondition>
86                <decomposition operator="ord">
87                    <activity name="aStep1FixBrokenConnector">
88                        <precondition>iPermanentlyAttachedConnector = Broken AND iPage = 13</
                              precondition>
89                        <decomposition operator="ord">
90                            <activity name="aReassembleConnector">
91                                <completioncondition>iPermanentlyAttachedConnector = Assembled</
                                      completioncondition>
92                                <decomposition operator="ord">
93                                    <action humanaction="hReassembleBrokenConnector"/>
94                                </decomposition>
95                            </activity>
96                            <activity name="aTryRotatingParts">
97                                <precondition>iRotationCounter = 0</precondition>
98                                <repeatcondition>iRotationCounter &lt; 3</repeatcondition>
99                                <completioncondition>iRotationCounter = 3 AND
                                      iPermanentlyAttachedConnector = Assembled</
                                      completioncondition>
100                               <decomposition operator="ord">
101                                   <action humanaction="hDisassembleConnector"/>
102                                   <action humanaction="hRotateConnectorParts"/>
103                                   <action humanaction="hReassembleBrokenConnector"/>
104                               </decomposition>
105                           </activity>
106                       </decomposition>
107                   </activity>
108                   <activity name="aStep2DisconnectOldParts">
109                       <precondition>iPage = 13</precondition>
110                       <decomposition operator="ord">
111                           <activity name="aDiscPumpFromOldController">
112                               <decomposition operator="xor">
113                                   <activity name="aDiscPumpCableFromAbCable">
114                                       <precondition>iPumpCableToOldAbCable = Connected AND
                                              iAbCableToOldController = Connected</precondition>
115                                       <decomposition operator="ord">
116                                           <action humanaction="hDiscPumpCableFromAbCable"/>
117                                       </decomposition>
118                                   </activity>
119                                   <activity name="aDiscPumpCableFromOldController">
120                                       <precondition>iPumpCableToOldController = Connected AND
                                              iAbCableToOldController = Disconnected</precondition
                                              >
121                                       <decomposition operator="ord">
122                                           <action humanaction="hDiscPumpCableFromOldController"
                                                  />
123                                       </decomposition>
124                                   </activity>
125                               </decomposition>
126                           </activity>
127                           <activity name="aSilenceAlarmOnOldController">
128                               <precondition>iOldControllerAlarmBatteryCap = Tightened</
                                      precondition>
129                               <completioncondition>iOldControllerAlarmBatteryCap = Loosened</
                                      completioncondition>
```

```
130                             <decomposition operator="ord">
131                                 <action humanaction="hLoosenOldControllerABCap"/>
132                             </decomposition>
133                         </activity>
134                         <activity name="aSetAsideOldComponents">
135                             <precondition>iOldComponents = AtHand</precondition>
136                             <completioncondition>iOldComponents = SetAside</
                                    completioncondition>
137                             <decomposition operator="ord">
138                                 <action humanaction="hSetAsideOldComponents"/>
139                             </decomposition>
140                          </activity>
141                     </decomposition>
142                 </activity>
143                 <activity name="aStep3ConnectNewController">
144                     <precondition>iPumpCableToOldController = Disconnected AND
                            iPumpCableToOldAbCable = Disconnected AND iPage = 13</precondition>
145                     <completioncondition>iPumpCableToNewController = Connected AND
                            iNewControllerAlarmBatteryCap = Tightened</completioncondition>
146                     <decomposition operator="ord">
147                         <activity name="aConPumpCableToNewController">
148                             <decomposition operator="ord">
149                                 <action humanaction="hConPumpCableToNewController"/>
150                             </decomposition>
151                         </activity>
152                         <activity name="aActivateAlarmOnNewController">
153                             <precondition>iNewControllerAlarmBatteryCap = Loosened</
                                    precondition>
154                             <completioncondition>iNewControllerAlarmBatteryCap = Tightened</
                                    completioncondition>
155                             <decomposition operator="ord">
156                                 <action humanaction="hTightenNewControllerABCap"/>
157                             </decomposition>
158                         </activity>
159                     </decomposition>
160                 </activity>
161                 <activity name="aStep4TryLeadBattery">
162                     <precondition>iPage = 13 AND iLeadBattToOldController = Disconnected AND
163                         NOT(iLeadBattToYCable = Connected AND iYCableToOldController =
                                Connected)</precondition>
164                     <decomposition operator="ord">
165                         <activity name="aConLeadBattToNewController">
166                             <precondition>iLeadBattToNewController = Disconnected</
                                    precondition>
167                             <completioncondition>iLeadBattToNewController = Connected</
                                    completioncondition>
168                             <decomposition operator="ord">
169                                 <action humanaction="hConLeadBattToNewController"/>
170                             </decomposition>
171                         </activity>
172                         <activity name="aDiscLeadBattFromNewController">
173                             <precondition>iLeadBattToNewController = Connected</precondition>
174                             <completioncondition>iLeadBattToNewController = Disconnected</
                                    completioncondition>
175                             <decomposition operator="ord">
176                                 <action humanaction="hDiscLeadBattFromNewController"/>
177                             </decomposition>
178                         </activity>
179                     </decomposition>
180                 </activity>
181                 <activity name="aStep5TryLiIonBattery">
182                     <precondition>iPage = 14</precondition>
183                     <decomposition operator="ord">
184                         <activity name="aCheckLiBatteryLevel">
185                             <decomposition operator="ord">
186                                 <action humanaction="hDepressBlackButtonOnNewLiBatt"/>
187                             </decomposition>
188                         </activity>
189                         <activity name="aConnectBattOrCallEmergencyNumber">
190                             <decomposition operator="xor">
191                                 <activity name="aCallEmergencyNumber">
192                                     <precondition>iNewLiBatteryLights &lt; 5</precondition>
193                                     <decomposition operator="ord">
194                                         <action humanaction="hCallEmergencyNumber"/>
195                                     </decomposition>
196                                 </activity>
197                                 <activity name="aConnectFullyChargedLiBatt">
198                                     <precondition>iNewLiBatteryLights = 5</precondition>
199                                     <decomposition operator="ord">
200                                         <activity name="aConNewLiBattCableToNewController">
```

```
201                                                 <precondition>iNewLiBattCableToNewController =
                                                        Disconnected</precondition>
202                                                 <completioncondition>
                                                        iNewLiBattCableToNewController = Connected</
                                                        completioncondition>
203                                                 <decomposition operator="ord">
204                                                     <action humanaction="
                                                            hConNewLiBattCableToNewController"/>
205                                                 </decomposition>
206                                             </activity>
207                                             <activity name="aConNewLiBattCableToNewLiBatt">
208                                                 <precondition>iNewLiBattCableToNewLiBatt =
                                                        Disconnected</precondition>
209                                                 <completioncondition>iNewLiBattCableToNewLiBatt =
                                                         Connected</completioncondition>
210                                                 <decomposition operator="ord">
211                                                     <action humanaction="
                                                            hConNewLiBattCableToNewLiBatt"/>
212                                                 </decomposition>
213                                             </activity>
214                                             <activity name="aBreakCircuit">
215                                                 <completioncondition>iPowerLight = 0</
                                                        completioncondition>
216                                                 <decomposition operator="ord">
217                                                     <action humanaction="
                                                            hDiscNewLiBattCableFromNewController"/>
218                                                 </decomposition>
219                                             </activity>
220                                             <activity name="aCallEmergency">
221                                                 <decomposition operator="ord">
222                                                     <action humanaction="hCallEmergencyNumber"/>
223                                                 </decomposition>
224                                             </activity>
225                                         </decomposition>
226                                     </activity>
227                                 </decomposition>
228                             </activity>
229                         </decomposition>
230                     </activity>
231                     <activity name="aStep6TagOldComponents">
232                         <precondition>iPage = 14 AND iOldComponentTags = notRedTagged</
                                precondition>
233                         <completioncondition>iOldComponentTags = redTagged</completioncondition>
234                         <decomposition operator="ord">
235                             <action humanaction="hRedTagOldComponents"/>
236                         </decomposition>
237                     </activity>
238                 </decomposition>
239             </activity>
240         </eofm>
241     </humanoperator>
242 </eofms>
```

H.1.2 Affordance Model

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <hes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:noNamespaceSchemaLocation="../../schema/cavemen.xsd">
4      <modelobject name="mOldController">
5          <atomicobject name="aoOCPumpInput"/>
6          <atomicobject name="aoOCBatteryInput"/>
7      </modelobject>
8      <modelobject name="mOldLiIonBattery">
9          <atomicobject name="aoOBBattCableInput"/>
10     </modelobject>
11     <modelobject name="mOldLiBattCable">
12         <atomicobject name="aoOBCBatteryOutput"/>
13         <atomicobject name="aoOBCControllerOutput"/>
14     </modelobject>
15     <modelobject name="mNewController">
16         <atomicobject name="aoNCPumpInput"/>
17         <atomicobject name="aoNCBatteryInput"/>
18     </modelobject>
19     <modelobject name="mNewLiIonBattery">
20         <atomicobject name="aoNBBattCableInput"/>
21     </modelobject>
22     <modelobject name="mNewLiBattCable">
23         <atomicobject name="aoNBCBatteryOutput"/>
```

```
24            <atomicobject name="aoNBCControllerOutput"/>
25        </modelobject>
26        <modelobject name="mLeadBattery">
27            <subobject name="sLeadBattCable">
28                <atomicobject name="aoLeadBattControllerOutput"/>
29            </subobject>
30        </modelobject>
31        <modelobject name="mPumpCable">
32            <subobject name="sConnector">
33                <atomicobject name="aoConnectorPart1"/>
34                <atomicobject name="aoConnectorPart2"/>
35            </subobject>
36            <atomicobject name="aoPCControllerOutput"/>
37        </modelobject>
38        <modelobject name="mAbdominalCable">
39            <atomicobject name="aoACControllerOutput"/>
40            <atomicobject name="aoACPumpInput"/>
41        </modelobject>
42        <modelobject name="mYCable">
43            <atomicobject name="aoYCControllerOutput"/>
44            <atomicobject name="aoYCableBatteryInput1"/>
45            <atomicobject name="aoYCableBatteryInput2"/>
46        </modelobject>
47
48        <affordance name="ConnectorPartsAssemblable" formalism="chemero">
49            <humanoperator     name="pPumpOperator">
50                <atomcomponent name="aoConnectorPart1">
51                    <relation  condition="not"      topology="covering"
52                               direction="front_of" associate="aoConnectorPart2"/>
53                    <ability   name="AssemblePart1">
54                        <positionable back="true"/>
55                    </ability>
56                </atomcomponent>
57                <atomcomponent name="aoConnectorPart2">
58                    <ability   name="AssemblePart2">
59                        <positionable forth="true"/>
60                    </ability>
61                </atomcomponent>
62            </humanoperator>
63        </affordance>
64        <affordance name="ConnectorPartsDisassemblable" formalism="chemero">
65            <humanoperator     name="pPumpOperator">
66                <atomcomponent name="aoConnectorPart1">
67                    <relation  topology="covering" direction="front_of"
68                               associate="aoConnectorPart2"/>
69                    <ability   name="DisassemblePart1">
70                        <positionable forth="true"/>
71                    </ability>
72                </atomcomponent>
73                <atomcomponent name="aoConnectorPart2">
74                    <ability   name="DisassemblePart2">
75                        <positionable back="true"/>
76                    </ability>
77                </atomcomponent>
78            </humanoperator>
79        </affordance>
80        <affordance name="ConnectorPartsRotatable" formalism="chemero">
81            <humanoperator     name="pPumpOperator">
82                <atomcomponent name="aoConnectorPart1">
83                    <relation  topology="disjoint_to" direction="front_of"
84                               associate="aoConnectorPart2"/>
85                    <ability   name="RotatePart1">
86                        <orientable roll_right="true" roll_left="true"/>
87                    </ability>
88                </atomcomponent>
89                <atomcomponent name="aoConnectorPart2">
90                    <relation  topology="disjoint_to" direction="back_of"
91                               associate="aoConnectorPart1"/>
92                    <ability   name="RotatePart2">
93                        <orientable roll_right="true" roll_left="true"/>
94                    </ability>
95                </atomcomponent>
96            </humanoperator>
97        </affordance>
98        <affordance name="PumpCableDisconnectableFromAbCable" formalism="chemero">
99            <humanoperator name="pPumpOperator">
100               <atomcomponent name="aoACPumpInput">
101                   <relation topology="covering" associate="aoPCControllerOutput"/>
102                   <ability name="MoveAbCablePumpInputBack">
103                       <positionable back="true"/>
104                   </ability>
```

```
105                 </atomcomponent>
106                 <atomcomponent name="aoPCControllerOutput">
107                     <ability name="MovePumpCableControllerOutputBack">
108                         <positionable back="true"/>
109                     </ability>
110                 </atomcomponent>
111             </humanoperator>
112         </affordance>
113         <affordance name="PumpCableDisconnectableFromOldController" formalism="chemero">
114             <humanoperator name="pPumpOperator">
115                 <component name="mOldController">
116                     <ability name="MoveOldControllerBack">
117                         <positionable     back="true"/>
118                     </ability>
119                 </component>
120                 <atomcomponent name="aoOCPumpInput">
121                     <relation topology="covering" associate="aoPCControllerOutput"/>
122                 </atomcomponent>
123                 <atomcomponent name="aoPCControllerOutput">
124                     <ability name="MovePumpCableControllerOutputBack">
125                         <positionable back="true"/>
126                     </ability>
127                 </atomcomponent>
128             </humanoperator>
129         </affordance>
130         <affordance          name="LeadBattDisconnectableFromNewController" formalism="chemero">
131             <humanoperator   name="pPumpOperator">
132                 <component   name="mNewController">
133                     <ability name="MoveNewControllerBack">
134                         <positionable back="true"/>
135                     </ability>
136                 </component>
137                 <atomcomponent name="aoNCBatteryInput">
138                     <relation  topology="covering" associate="aoLeadBattControllerOutput"/>
139                 </atomcomponent>
140                 <atomcomponent name="aoLeadBattControllerOutput">
141                     <ability   name="MoveLeadBattCableControllerOutputBack">
142                         <positionable back="true"/>
143                     </ability>
144                 </atomcomponent>
145             </humanoperator>
146         </affordance>
147         <affordance name="NewLiBattCableDisconnectableFromNewController" formalism="chemero">
148             <humanoperator   name="pPumpOperator">
149                 <component   name="mNewController">
150                     <ability name="MoveNewControllerBack">
151                         <positionable back="true"/>
152                     </ability>
153                 </component>
154                 <atomcomponent name="aoNCBatteryInput">
155                     <relation  topology="covering" associate="aoNBCControllerOutput"/>
156                 </atomcomponent>
157                 <atomcomponent name="aoNBCControllerOutput">
158                     <ability   name="MoveNewLiBattCableControllerOutputBack">
159                         <positionable back="true"/>
160                     </ability>
161                 </atomcomponent>
162             </humanoperator>
163         </affordance>
164         <affordance name="PumpCableConnectableToNewController" formalism="chemero">
165             <humanoperator name="pPumpOperator">
166                 <component name="mNewController">
167                     <ability name="MoveNewController">
168                         <orientable roll_left="true" roll_right="true"
169                             pitch_back="true" pitch_forth="true"
170                             yaw_left="true"   yaw_right="true"/>
171                         <translatable     left="true"       right="true"/>
172                         <positionable     back="true"       forth="true"
173                             up="true"        down="true"/>
174                     </ability>
175                 </component>
176                 <atomcomponent name="aoOCPumpInput">
177                     <relation condition="not" topology="covering"
178                               associate="aoPCControllerOutput"/>
179                 </atomcomponent>
180                 <atomcomponent name="aoNCPumpInput">
181                     <relation condition="not" topology="covering"
182                               associate="aoPCControllerOutput"/>
183                 </atomcomponent>
184                 <atomcomponent name="aoACPumpInput">
185                     <relation condition="not" topology="covering"
```

```
186                           associate="aoPCControllerOutput"/>
187             </atomcomponent>
188             <atomcomponent name="aoPCControllerOutput">
189                 <ability name="MovePumpCableOutput">
190                     <orientable pitch_back="true" pitch_forth="true"
191                         yaw_left="true"   yaw_right="true"/>
192                     <translatable     left="true"       right="true"/>
193                     <positionable     back="true"       forth="true"
194                         up="true"        down="true"/>
195                 </ability>
196             </atomcomponent>
197         </humanoperator>
198     </affordance>
199     <affordance          name="LeadBattConnectableToNewController" formalism="chemero">
200         <humanoperator   name="pPumpOperator">
201             <component   name="mNewController">
202                 <ability name="MoveNewController">
203                     <orientable roll_left="true" roll_right="true"
204                                 pitch_back="true" pitch_forth="true"
205                                 yaw_left="true"   yaw_right="true"/>
206                     <translatable left="true" right="true"/>
207                     <positionable back="true" forth="true"
208                                   up="true"   down="true"/>
209                 </ability>
210             </component>
211             <atomcomponent name="aoNCBatteryInput">
212                 <relation  condition="not" topology="covering"
213                            associate="aoLeadBattControllerOutput"/>
214                 <relation  condition="not" topology="covering"
215                            associate="aoOBCControllerOutput"/>
216                 <relation  condition="not" topology="covering"
217                            associate="aoNBCControllerOutput"/>
218                 <relation  condition="not" topology="covering"
219                            associate="aoYCControllerOutput"/>
220             </atomcomponent>
221             <atomcomponent name="aoOCBatteryInput">
222                 <relation  condition="not" topology="covering"
223                            associate="aoLeadBattControllerOutput"/>
224             </atomcomponent>
225             <atomcomponent name="aoYCableBatteryInput1">
226                 <relation  condition="not" topology="covering"
227                            associate="aoLeadBattControllerOutput"/>
228             </atomcomponent>
229             <atomcomponent name="aoYCableBatteryInput2">
230                 <relation  condition="not" topology="covering"
231                            associate="aoLeadBattControllerOutput"/>
232             </atomcomponent>
233             <atomcomponent name="aoLeadBattControllerOutput">
234                 <ability   name="MoveLeadBattControllerOutput">
235                     <orientable pitch_back="true" pitch_forth="true"
236                                 yaw_left="true"   yaw_right="true"/>
237                     <translatable left="true" right="true"/>
238                     <positionable back="true" forth="true"
239                                   up="true"   down="true"/>
240                 </ability>
241             </atomcomponent>
242         </humanoperator>
243     </affordance>
244     <affordance          name="NewLiBattCableConnectableToNewController" formalism="chemero">
245         <humanoperator   name="pPumpOperator">
246             <component   name="mNewController">
247                 <ability name="MoveNewController">
248                     <orientable roll_left="true" roll_right="true"
249                         pitch_back="true" pitch_forth="true"
250                         yaw_left="true"   yaw_right="true"/>
251                     <translatable     left="true"       right="true"/>
252                     <positionable     back="true"       forth="true"
253                         up="true"        down="true"/>
254                 </ability>
255             </component>
256             <atomcomponent name="aoNCBatteryInput">
257                 <relation condition="not" topology="covering"
258                           associate="aoOBCControllerOutput"/>
259                 <relation condition="not" topology="covering"
260                           associate="aoNBCControllerOutput"/>
261                 <relation condition="not" topology="covering"
262                           associate="aoYCControllerOutput"/>
263                 <relation condition="not" topology="covering"
264                           associate="aoLeadBattControllerOutput"/>
265             </atomcomponent>
266             <atomcomponent name="aoOCBatteryInput">
```

```
267                     <relation condition="not" topology="covering"
268                               associate="aoNBCControllerOutput"/>
269                 </atomcomponent>
270                 <atomcomponent name="aoYCableBatteryInput1">
271                     <relation condition="not" topology="covering"
272                               associate="aoNBCControllerOutput"/>
273                 </atomcomponent>
274                 <atomcomponent name="aoYCableBatteryInput2">
275                     <relation condition="not" topology="covering"
276                               associate="aoNBCControllerOutput"/>
277                 </atomcomponent>
278                 <atomcomponent name="aoNBCControllerOutput">
279                     <ability name="MoveNewBattCableControllerOutput">
280                         <orientable pitch_back="true" pitch_forth="true"
281                                     yaw_left="true"   yaw_right="true"/>
282                         <translatable     left="true"       right="true"/>
283                         <positionable     back="true"       forth="true"
284                                           up="true"        down="true"/>
285                     </ability>
286                 </atomcomponent>
287             </humanoperator>
288         </affordance>
289         <affordance name="OldLiBattCableConnectableToNewController" formalism="chemero">
290             <humanoperator name="pPumpOperator">
291                 <component name="mNewController">
292                     <ability name="MoveNewController">
293                         <orientable roll_left="true" roll_right="true"
294                             pitch_back="true" pitch_forth="true"
295                             yaw_left="true"   yaw_right="true"/>
296                         <translatable     left="true"       right="true"/>
297                         <positionable     back="true"       forth="true"
298                             up="true"        down="true"/>
299                     </ability>
300                 </component>
301                 <atomcomponent name="aoNCBatteryInput">
302                     <relation condition="not" topology="covering"
303                               associate="aoOBCControllerOutput"/>
304                     <relation condition="not" topology="covering"
305                               associate="aoNBCControllerOutput"/>
306                     <relation condition="not" topology="covering"
307                               associate="aoYCControllerOutput"/>
308                     <relation condition="not" topology="covering"
309                               associate="aoLeadBattControllerOutput"/>
310                 </atomcomponent>
311                 <atomcomponent name="aoYCableBatteryInput1">
312                     <relation condition="not" topology="covering"
313                               associate="aoOBCControllerOutput"/>
314                 </atomcomponent>
315                 <atomcomponent name="aoYCableBatteryInput2">
316                     <relation condition="not" topology="covering"
317                               associate="aoOBCControllerOutput"/>
318                 </atomcomponent>
319                 <atomcomponent name="aoOCBatteryInput">
320                     <relation condition="not" topology="covering"
321                               associate="aoOBCControllerOutput"/>
322                 </atomcomponent>
323                 <atomcomponent name="aoOBCControllerOutput">
324                     <ability name="MoveOldBattCableControllerOutput">
325                         <orientable pitch_back="true" pitch_forth="true"
326                             yaw_left="true"   yaw_right="true"/>
327                         <translatable     left="true"       right="true"/>
328                         <positionable     back="true"       forth="true"
329                             up="true"        down="true"/>
330                     </ability>
331                 </atomcomponent>
332             </humanoperator>
333         </affordance>
334         <affordance          name="NewLiBattCableConnectableToNewLiBatt" formalism="chemero">
335             <humanoperator   name="pPumpOperator">
336                 <component   name="mNewLiIonBattery">
337                     <ability name="MoveNewLiBattery">
338                         <orientable pitch_back="true" pitch_forth="true"
339                                     yaw_left="true"   yaw_right="true"/>
340                         <translatable left="true" right="true"/>
341                         <positionable back="true" forth="true"
342                                       up="true"   down="true"/>
343                     </ability>
344                 </component>
345                 <atomcomponent name="aoOBBattCableInput">
346                     <relation  condition="not" topology="covering"
347                                associate="aoNBCBatteryOutput"/>
```

```
348             </atomcomponent>
349             <atomcomponent name="aoNBBattCableInput">
350                 <relation  condition="not" topology="covering"
351                            associate="aoNBCBatteryOutput"/>
352                 <relation  condition="not" topology="covering"
353                            associate="aoOBCBatteryOutput"/>
354             </atomcomponent>
355             <atomcomponent name="aoNBCBatteryOutput">
356                 <ability   name="MoveNewBattCableBatteryOutput">
357                     <orientable pitch_back="true" pitch_forth="true"
358                                 yaw_left="true"   yaw_right="true"/>
359                     <translatable left="true" right="true"/>
360                     <positionable back="true" forth="true"
361                                   up="true"   down="true"/>
362                 </ability>
363             </atomcomponent>
364         </humanoperator>
365     </affordance>
366     <affordance          name="NewLiBattCableConnectableToOldLiBatt" formalism="chemero">
367         <humanoperator   name="pPumpOperator">
368             <component   name="mOldLiIonBattery">
369                 <ability name="MoveOldLiBattery">
370                     <orientable pitch_back="true" pitch_forth="true"
371                                 yaw_left="true"   yaw_right="true"/>
372                     <translatable left="true" right="true"/>
373                     <positionable back="true" forth="true"
374                                   up="true"   down="true"/>
375                 </ability>
376             </component>
377             <atomcomponent name="aoOBBattCableInput">
378                 <relation  condition="not" topology="covering"
379                            associate="aoNBCBatteryOutput"/>
380                 <relation  condition="not" topology="covering"
381                            associate="aoOBCBatteryOutput"/>
382             </atomcomponent>
383             <atomcomponent name="aoNBBattCableInput">
384                 <relation  condition="not" topology="covering"
385                            associate="aoNBCBatteryOutput"/>
386             </atomcomponent>
387             <atomcomponent name="aoNBCBatteryOutput">
388                 <ability   name="MoveNewBattCableBatteryOutput">
389                     <orientable pitch_back="true" pitch_forth="true"
390                                 yaw_left="true"   yaw_right="true"/>
391                     <translatable left="true" right="true"/>
392                     <positionable back="true" forth="true"
393                                   up="true"   down="true"/>
394                 </ability>
395             </atomcomponent>
396         </humanoperator>
397     </affordance>
398 </hes>
```

H.1.3 Signifier Model

```
1  <bigsis xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2      xsi:noNamespaceSchemaLocation="../../schema/bigsis-2.0.xsd">
3      <signified-meanings name="PumpSpeed">Stopped, Low, Lowest, Medium, High, Highest</signified-
           meanings>
4      <signified-meanings name="PowerSupplied">ZeroUnits, OneUnit, TwoUnits, ThreeUnits, FourUnits,
5          FiveUnits, SixUnits, SevenUnits, EightUnits, NineUnits, TenUnits, TooHigh</signified-
               meanings>
6      <signifier-properties of="PowerIndicators">
7          <Color signifies="PowerSupplied" when-colored="green">PowerIndicators.Label.PowerSupplied
               </Color>
8          <Color signifies="PowerSupplied" when-colored="amber">TooHigh</Color>
9          <Label signifies="PowerSupplied" when-labeled="one">OneUnit</Label>
10         <Label signifies="PowerSupplied" when-labeled="two">TwoUnits</Label>
11         <Label signifies="PowerSupplied" when-labeled="three">ThreeUnits</Label>
12         <Label signifies="PowerSupplied" when-labeled="four">FourUnits</Label>
13         <Label signifies="PowerSupplied" when-labeled="five">FiveUnits</Label>
14         <Label signifies="PowerSupplied" when-labeled="six">SixUnits</Label>
15         <Label signifies="PowerSupplied" when-labeled="seven">SevenUnits</Label>
16         <Label signifies="PowerSupplied" when-labeled="eight">EightUnits</Label>
17         <Label signifies="PowerSupplied" when-labeled="nine">NineUnits</Label>
18         <Label signifies="PowerSupplied" when-labeled="ten">TenUnits</Label>
19         <Label signifies="PowerSupplied" when-labeled="HIGH">TooHigh</Label>
20         <Volume signifies="PowerSupplied" when-level="loud">PowerIndicators.aPattern.
               PowerSupplied</Volume>
21         <aPattern signifies="PowerSupplied" when-pattern="POWER_TOO_HIGH">TooHigh</aPattern>
22     </signifier-properties>
```

```
    <signifier-properties of="PumpStoppedAlarm">
        <Color signifies="PumpSpeed" when-colored="red">Stopped</Color>
        <Color signifies="PumpSpeed" when-colored="noColor">SpeedSettingKnob.Label.PumpSpeed</Color>
        <Volume signifies="PumpSpeed" when-level="loud">PumpStoppedAlarm.aPattern.PumpSpeed</Volume>
        <aPattern signifies="PumpSpeed" when-pattern="PUMP_STOPPED">Stopped</aPattern>
    </signifier-properties>
    <signifier-properties of="SpeedSettingKnob">
        <Color signifies="PumpSpeed" when-colored="white">SpeedSettingKnob.Label.PumpSpeed</Color>
        <Color signifies="PumpSpeed" when-colored="noColor">PumpStoppedAlarm.Color.PumpSpeed</Color>
        <Label signifies="PumpSpeed" when-labeled="noLabel">PumpStoppedAlarm.Color.PumpSpeed</Label>
        <Label signifies="PumpSpeed" when-labeled="one">Lowest</Label>
        <Label signifies="PumpSpeed" when-labeled="two">Low</Label>
        <Label signifies="PumpSpeed" when-labeled="three">Medium</Label>
        <Label signifies="PumpSpeed" when-labeled="four">High</Label>
        <Label signifies="PumpSpeed" when-labeled="five">Highest</Label>
    </signifier-properties>
    <property-documentation of="PowerIndicators">
        <Color signifies="PowerSupplied" when-colored="green">PowerIndicators.Label.PowerSupplied</Color>
        <Color signifies="PowerSupplied" when-colored="noColor">ZeroUnits</Color>
        <Color signifies="PumpSpeed" when-colored="noColor">Stopped</Color>
        <Color signifies="PowerSupplied" when-colored="noColor">ZeroUnits</Color>
        <Label signifies="PowerSupplied" when-labeled="noLabel">PowerIndicators.Color.PowerSupplied</Label>
        <Label signifies="PowerSupplied" when-labeled="HIGH">TooHigh</Label>
        <Label signifies="PowerSupplied" when-labeled="one">OneUnit</Label>
        <Label signifies="PowerSupplied" when-labeled="two">TwoUnits</Label>
        <Label signifies="PowerSupplied" when-labeled="three">ThreeUnits</Label>
        <Label signifies="PowerSupplied" when-labeled="four">FourUnits</Label>
        <Label signifies="PowerSupplied" when-labeled="five">FiveUnits</Label>
        <Label signifies="PowerSupplied" when-labeled="six">SixUnits</Label>
        <Label signifies="PowerSupplied" when-labeled="seven">SevenUnits</Label>
        <Label signifies="PowerSupplied" when-labeled="eight">EightUnits</Label>
        <Label signifies="PowerSupplied" when-labeled="nine">NineUnits</Label>
        <Label signifies="PowerSupplied" when-labeled="ten">TenUnits</Label>
        <Color signifies="PowerSupplied" when-colored="amber">TooHigh</Color>
        <Volume signifies="PowerSupplied" when-level="loud">PowerIndicators.aPattern.PowerSupplied</Volume>
        <aPattern signifies="PowerSupplied" when-pattern="POWER_TOO_HIGH">TooHigh</aPattern>
    </property-documentation>
    <property-documentation of="PumpStoppedAlarm">
        <Color signifies="PumpSpeed" when-colored="red">Stopped</Color>
        <Color signifies="PumpSpeed" when-colored="noColor">SpeedSettingKnob.Label.PumpSpeed</Color>
        <Volume signifies="PumpSpeed" when-level="loud">PumpStoppedAlarm.aPattern.PumpSpeed</Volume>
        <aPattern signifies="PumpSpeed" when-pattern="PUMP_STOPPED">Stopped</aPattern>
    </property-documentation>
    <property-documentation of="SpeedSettingKnob">
        <Color signifies="PumpSpeed" when-colored="white">SpeedSettingKnob.Label.PumpSpeed</Color>
        <Color signifies="PumpSpeed" when-colored="noColor">PumpStoppedAlarm.Color.PumpSpeed</Color>
        <Label signifies="PumpSpeed" when-labeled="noLabel">Stopped</Label>
        <Label signifies="PumpSpeed" when-labeled="one">Lowest</Label>
        <Label signifies="PumpSpeed" when-labeled="two">Low</Label>
        <Label signifies="PumpSpeed" when-labeled="three">Medium</Label>
        <Label signifies="PumpSpeed" when-labeled="four">High</Label>
        <Label signifies="PumpSpeed" when-labeled="five">Highest</Label>
    </property-documentation>
</bigsis>
```

H.2 SAL Code

H.2.1 Documentation Navigation

```
documentation: CONTEXT =
BEGIN
  keepPage(iPage: INTEGER): INTEGER = iPage;
  turnPage(iPage: INTEGER): INTEGER = iPage + 1;
  crossRef(ref: INTEGER): INTEGER = ref;
  navigation: MODULE =
  BEGIN
    OUTPUT iPage: {x: INTEGER | x >= 0 AND x <= 29}
```

```
9       INITIALIZATION
10        iPage = 2;
11      TRANSITION
12      [
13      iPage  = 2  -->
14        iPage' IN {crossRef(8), crossRef(10), crossRef(13), crossRef(15)};
15      []iPage  = 8 -->
16        iPage' IN {keepPage(iPage), crossRef(2), crossRef(10)};
17      []iPage  = 10 -->
18        iPage' IN {keepPage(iPage), crossRef(2), crossRef(8), crossRef(13), crossRef(15)};
19      []iPage = 13 -->
20        iPage' IN {keepPage(iPage), turnPage(iPage), crossRef(2), crossRef(10)};
21      []iPage = 14 -->
22        iPage' IN {keepPage(iPage), crossRef(2)};
23      []iPage = 15 -->
24        iPage' IN {keepPage(iPage), crossRef(2)};
25    ]
26    END;
27  END
```

H.2.2 Task Model

```
1   userManual : CONTEXT =
2   BEGIN
3
4     tActivityState: TYPE = {actReady, actExecuting, actDone};
5     tRotationCounter: TYPE = [0..3];
6     tOldComponentsLocation: TYPE = {AtHand, SetAside};
7     tConnection: TYPE = {Connected, Disconnected};
8     tPartTag: TYPE = {redTagged, notRedTagged};
9     tPermAttachedConnectorStatus: TYPE = {Broken, Assembled};
10    tAlarmBatteryCap: TYPE = {Loosened, Tightened};
11
12    system: MODULE =
13    BEGIN
14      OUTPUT ready: BOOLEAN
15      INPUT submitted: BOOLEAN
16
17      INITIALIZATION
18        ready = FALSE;
19
20      TRANSITION
21      [
22        NOT (ready OR submitted) -->
23          ready' = TRUE;
24        []ready AND submitted -->
25          ready' = FALSE;
26      ];
27    END;
28
29    humanOperators: MODULE =
30    BEGIN
31      %% Variables for pPumpOperator
32      INPUT iPage: {x: INTEGER | x >= 0 AND x <= 29}
33      INPUT iAlarm: discreteDevice!Alarms
34      INPUT iSpeedSetting: discreteDevice!SpeedSettings
35      INPUT iPowerLight: discreteDevice!PowerLights
36      INPUT iOldComponentTags: tPartTag
37      INPUT iLeadBattToOldController: tConnection
38      INPUT iLeadBattToNewController: tConnection
39      INPUT iLeadBattToYCable: tConnection
40      INPUT iYCableToOldController: tConnection
41      INPUT iOldLiBattCableToYCable: tConnection
42      INPUT iOldLiBattCableToOldController: tConnection
43      INPUT iOldLiBattCableToOldLiBatt: tConnection
44      INPUT iPumpCableToOldController: tConnection
45      INPUT iAbCableToOldController: tConnection
46      INPUT iNewLiBattCableToNewLiBatt: tConnection
47      INPUT iNewLiBattCableToNewController: tConnection
48      INPUT iPumpCableToOldAbCable: tConnection
49      INPUT iPumpCableToNewController: tConnection
50      INPUT iNewLiBatteryLights: discreteDevice!BatteryLights
51      INPUT iPermanentlyAttachedConnector: tPermAttachedConnectorStatus
52      INPUT iRotationCounter: tRotationCounter
53      INPUT iNewControllerABCap: tAlarmBatteryCap
54      INPUT iOldControllerABCap: tAlarmBatteryCap
55      INPUT iOldComponents: tOldComponentsLocation
56      OUTPUT hCallEmergencyNumber: BOOLEAN
57      OUTPUT hRedTagOldComponents: BOOLEAN
58      OUTPUT hSetAsideOldComponents: BOOLEAN
```

```
59      OUTPUT hRotateConnectorParts: BOOLEAN
60      OUTPUT hDisassembleConnector: BOOLEAN
61      OUTPUT hReassembleBrokenConnector: BOOLEAN
62      OUTPUT hDiscPumpCableFromAbCable: BOOLEAN
63      OUTPUT hDiscPumpCableFromOldController: BOOLEAN
64      OUTPUT hDiscLeadBattFromNewController: BOOLEAN
65      OUTPUT hDiscNewLiBattCableFromNewController: BOOLEAN
66      OUTPUT hConNewLiBattCableToNewController: BOOLEAN
67      OUTPUT hConNewLiBattCableToNewLiBatt: BOOLEAN
68      OUTPUT hDiscNewLiBattCableFromNewLiBatt: BOOLEAN
69      OUTPUT hConLeadBattToNewController: BOOLEAN
70      OUTPUT hConPumpCableToNewController: BOOLEAN
71      OUTPUT hConNewLeadBattToNewController: BOOLEAN
72      OUTPUT hDepressBlackButtonOnNewLiBatt: BOOLEAN
73      OUTPUT hRotateKnobClockwise: BOOLEAN
74      OUTPUT hRotateKnobCounterclockwise: BOOLEAN
75      OUTPUT hTightenNewControllerABCap: BOOLEAN
76      OUTPUT hLoosenOldControllerABCap: BOOLEAN
77      OUTPUT hLoosenNewControllerABCap: BOOLEAN
78      LOCAL lDesiredSpeed: discreteDevice!SpeedSettings
79      LOCAL aAdjustSpeed_Ready: BOOLEAN
80      LOCAL aAdjustSpeed_Executing: BOOLEAN
81      LOCAL aAdjustSpeed_Done: BOOLEAN
82      LOCAL aIncreaseSetting_Ready: BOOLEAN
83      LOCAL aIncreaseSetting_Executing: BOOLEAN
84      LOCAL aIncreaseSetting_Done: BOOLEAN
85      LOCAL aIncreaseSetting_Repeating: BOOLEAN
86      LOCAL hRotateKnobCounterclockwise_1: tActivityState
87      LOCAL aDecreaseSetting_Ready: BOOLEAN
88      LOCAL aDecreaseSetting_Executing: BOOLEAN
89      LOCAL aDecreaseSetting_Done: BOOLEAN
90      LOCAL aDecreaseSetting_Repeating: BOOLEAN
91      LOCAL hRotateKnobClockwise_2: tActivityState
92      LOCAL aRespondToPumpStoppedAlarm_Ready: BOOLEAN
93      LOCAL aRespondToPumpStoppedAlarm_Executing: BOOLEAN
94      LOCAL aRespondToPumpStoppedAlarm_Done: BOOLEAN
95      LOCAL aStep1FixBrokenConnector_Ready: BOOLEAN
96      LOCAL aStep1FixBrokenConnector_Executing: BOOLEAN
97      LOCAL aStep1FixBrokenConnector_Done: BOOLEAN
98      LOCAL aReassembleConnector_Ready: BOOLEAN
99      LOCAL aReassembleConnector_Executing: BOOLEAN
100     LOCAL aReassembleConnector_Done: BOOLEAN
101     LOCAL hReassembleBrokenConnector_3: tActivityState
102     LOCAL aTryRotatingParts_Ready: BOOLEAN
103     LOCAL aTryRotatingParts_Executing: BOOLEAN
104     LOCAL aTryRotatingParts_Done: BOOLEAN
105     LOCAL aTryRotatingParts_Repeating: BOOLEAN
106     LOCAL hDisassembleConnector_4: tActivityState
107     LOCAL hRotateConnectorParts_5: tActivityState
108     LOCAL hReassembleBrokenConnector_6: tActivityState
109     LOCAL aStep2DisconnectOldParts_Ready: BOOLEAN
110     LOCAL aStep2DisconnectOldParts_Executing: BOOLEAN
111     LOCAL aStep2DisconnectOldParts_Done: BOOLEAN
112     LOCAL aDiscPumpFromOldController_Ready: BOOLEAN
113     LOCAL aDiscPumpFromOldController_Executing: BOOLEAN
114     LOCAL aDiscPumpFromOldController_Done: BOOLEAN
115     LOCAL aDiscPumpCableFromAbCable_Ready: BOOLEAN
116     LOCAL aDiscPumpCableFromAbCable_Executing: BOOLEAN
117     LOCAL aDiscPumpCableFromAbCable_Done: BOOLEAN
118     LOCAL hDiscPumpCableFromAbCable_7: tActivityState
119     LOCAL aDiscPumpCableFromOldController_Ready: BOOLEAN
120     LOCAL aDiscPumpCableFromOldController_Executing: BOOLEAN
121     LOCAL aDiscPumpCableFromOldController_Done: BOOLEAN
122     LOCAL hDiscPumpCableFromOldController_8: tActivityState
123     LOCAL aSilenceAlarmOnOldController_Ready: BOOLEAN
124     LOCAL aSilenceAlarmOnOldController_Executing: BOOLEAN
125     LOCAL aSilenceAlarmOnOldController_Done: BOOLEAN
126     LOCAL hLoosenOldControllerABCap_9: tActivityState
127     LOCAL aSetAsideOldComponents_Ready: BOOLEAN
128     LOCAL aSetAsideOldComponents_Executing: BOOLEAN
129     LOCAL aSetAsideOldComponents_Done: BOOLEAN
130     LOCAL hSetAsideOldComponents_10: tActivityState
131     LOCAL aStep3ConnectNewController_Ready: BOOLEAN
132     LOCAL aStep3ConnectNewController_Executing: BOOLEAN
133     LOCAL aStep3ConnectNewController_Done: BOOLEAN
134     LOCAL aConPumpCableToNewController_Ready: BOOLEAN
135     LOCAL aConPumpCableToNewController_Executing: BOOLEAN
136     LOCAL aConPumpCableToNewController_Done: BOOLEAN
137     LOCAL hConPumpCableToNewController_11: tActivityState
138     LOCAL aActivateAlarmOnNewController_Ready: BOOLEAN
139     LOCAL aActivateAlarmOnNewController_Executing: BOOLEAN
```

```
140    LOCAL aActivateAlarmOnNewController_Done: BOOLEAN
141    LOCAL hTightenNewControllerABCap_12: tActivityState
142    LOCAL aStep4TryLeadBattery_Ready: BOOLEAN
143    LOCAL aStep4TryLeadBattery_Executing: BOOLEAN
144    LOCAL aStep4TryLeadBattery_Done: BOOLEAN
145    LOCAL aConLeadBattToNewController_Ready: BOOLEAN
146    LOCAL aConLeadBattToNewController_Executing: BOOLEAN
147    LOCAL aConLeadBattToNewController_Done: BOOLEAN
148    LOCAL hConLeadBattToNewController_13: tActivityState
149    LOCAL aDiscLeadBattFromNewController_Ready: BOOLEAN
150    LOCAL aDiscLeadBattFromNewController_Executing: BOOLEAN
151    LOCAL aDiscLeadBattFromNewController_Done: BOOLEAN
152    LOCAL hDiscLeadBattFromNewController_14: tActivityState
153    LOCAL aStep5TryLiIonBattery_Ready: BOOLEAN
154    LOCAL aStep5TryLiIonBattery_Executing: BOOLEAN
155    LOCAL aStep5TryLiIonBattery_Done: BOOLEAN
156    LOCAL aCheckLiBatteryLevel_Ready: BOOLEAN
157    LOCAL aCheckLiBatteryLevel_Executing: BOOLEAN
158    LOCAL aCheckLiBatteryLevel_Done: BOOLEAN
159    LOCAL hDepressBlackButtonOnNewLiBatt_15: tActivityState
160    LOCAL aConnectBattOrCallEmergencyNumber_Ready: BOOLEAN
161    LOCAL aConnectBattOrCallEmergencyNumber_Executing: BOOLEAN
162    LOCAL aConnectBattOrCallEmergencyNumber_Done: BOOLEAN
163    LOCAL aCallEmergencyNumber_Ready: BOOLEAN
164    LOCAL aCallEmergencyNumber_Executing: BOOLEAN
165    LOCAL aCallEmergencyNumber_Done: BOOLEAN
166    LOCAL hCallEmergencyNumber_16: tActivityState
167    LOCAL aConnectFullyChargedLiBatt_Ready: BOOLEAN
168    LOCAL aConnectFullyChargedLiBatt_Executing: BOOLEAN
169    LOCAL aConnectFullyChargedLiBatt_Done: BOOLEAN
170    LOCAL aConNewLiBattCableToNewController_Ready: BOOLEAN
171    LOCAL aConNewLiBattCableToNewController_Executing: BOOLEAN
172    LOCAL aConNewLiBattCableToNewController_Done: BOOLEAN
173    LOCAL hConNewLiBattCableToNewController_17: tActivityState
174    LOCAL aConNewLiBattCableToNewLiBatt_Ready: BOOLEAN
175    LOCAL aConNewLiBattCableToNewLiBatt_Executing: BOOLEAN
176    LOCAL aConNewLiBattCableToNewLiBatt_Done: BOOLEAN
177    LOCAL hConNewLiBattCableToNewLiBatt_18: tActivityState
178    LOCAL aBreakCircuit_Ready: BOOLEAN
179    LOCAL aBreakCircuit_Executing: BOOLEAN
180    LOCAL aBreakCircuit_Done: BOOLEAN
181    LOCAL hDiscNewLiBattCableFromNewController_19: tActivityState
182    LOCAL aCallEmergency_Ready: BOOLEAN
183    LOCAL aCallEmergency_Executing: BOOLEAN
184    LOCAL aCallEmergency_Done: BOOLEAN
185    LOCAL hCallEmergencyNumber_20: tActivityState
186    LOCAL aStep6TagOldComponents_Ready: BOOLEAN
187    LOCAL aStep6TagOldComponents_Executing: BOOLEAN
188    LOCAL aStep6TagOldComponents_Done: BOOLEAN
189    LOCAL hRedTagOldComponents_21: tActivityState
190
191    INITIALIZATION
192      hCallEmergencyNumber = FALSE;
193      hRedTagOldComponents = FALSE;
194      hSetAsideOldComponents = FALSE;
195      hRotateConnectorParts = FALSE;
196      hDisassembleConnector = FALSE;
197      hReassembleBrokenConnector = FALSE;
198      hDiscPumpCableFromAbCable = FALSE;
199      hDiscPumpCableFromOldController = FALSE;
200      hDiscLeadBattFromNewController = FALSE;
201      hDiscNewLiBattCableFromNewController = FALSE;
202      hConNewLiBattCableToNewController = FALSE;
203      hConNewLiBattCableToNewLiBatt = FALSE;
204      hDiscNewLiBattCableFromNewLiBatt = FALSE;
205      hConLeadBattToNewController = FALSE;
206      hConPumpCableToNewController = FALSE;
207      hConNewLeadBattToNewController = FALSE;
208      hDepressBlackButtonOnNewLiBatt = FALSE;
209      hRotateKnobClockwise = FALSE;
210      hRotateKnobCounterclockwise = FALSE;
211      hTightenNewControllerABCap = FALSE;
212      hLoosenOldControllerABCap = FALSE;
213      hLoosenNewControllerABCap = FALSE;
214      lDesiredSpeed IN {1, 2, 3, 4, 5};
215      aIncreaseSetting_Done = FALSE;
216      aIncreaseSetting_Repeating = FALSE;
217      hRotateKnobCounterclockwise_1 = actReady;
218      aDecreaseSetting_Done = FALSE;
219      aDecreaseSetting_Repeating = FALSE;
220      hRotateKnobClockwise_2 = actReady;
```

```
221       hReassembleBrokenConnector_3 = actReady;
222       aTryRotatingParts_Done = FALSE;
223       aTryRotatingParts_Repeating = FALSE;
224       hDisassembleConnector_4 = actReady;
225       hRotateConnectorParts_5 = actReady;
226       hReassembleBrokenConnector_6 = actReady;
227       hDiscPumpCableFromAbCable_7 = actReady;
228       hDiscPumpCableFromOldController_8 = actReady;
229       hLoosenOldControllerABCap_9 = actReady;
230       hSetAsideOldComponents_10 = actReady;
231       hConPumpCableToNewController_11 = actReady;
232       hTightenNewControllerABCap_12 = actReady;
233       hConLeadBattToNewController_13 = actReady;
234       hDiscLeadBattFromNewController_14 = actReady;
235       hDepressBlackButtonOnNewLiBatt_15 = actReady;
236       hCallEmergencyNumber_16 = actReady;
237       hConNewLiBattCableToNewController_17 = actReady;
238       hConNewLiBattCableToNewLiBatt_18 = actReady;
239       hDiscNewLiBattCableFromNewController_19 = actReady;
240       hCallEmergencyNumber_20 = actReady;
241       hRedTagOldComponents_21 = actReady;
242
243     %% Handshake variables
244     INPUT ready: BOOLEAN
245     OUTPUT submitted: BOOLEAN
246     INITIALIZATION
247       submitted = FALSE;
248
249
250     DEFINITION
251       aAdjustSpeed_Ready = (aIncreaseSetting_Ready) AND (aDecreaseSetting_Ready);
252       aAdjustSpeed_Executing = NOT (aAdjustSpeed_Ready) AND NOT (aAdjustSpeed_Done);
253       aAdjustSpeed_Done = (aIncreaseSetting_Done) AND (aDecreaseSetting_Done);
254       aIncreaseSetting_Ready = (hRotateKnobCounterclockwise_1 = actReady) AND (NOT
              aIncreaseSetting_Repeating);
255       aIncreaseSetting_Executing = NOT (aIncreaseSetting_Ready) AND NOT (aIncreaseSetting_Done);
256       aDecreaseSetting_Ready = (hRotateKnobClockwise_2 = actReady) AND (NOT
              aDecreaseSetting_Repeating);
257       aDecreaseSetting_Executing = NOT (aDecreaseSetting_Ready) AND NOT (aDecreaseSetting_Done);
258       aRespondToPumpStoppedAlarm_Ready = (aStep1FixBrokenConnector_Ready) AND (
              aStep2DisconnectOldParts_Ready) AND (aStep3ConnectNewController_Ready) AND (
              aStep4TryLeadBattery_Ready) AND (aStep5TryLiIonBattery_Ready) AND (
              aStep6TagOldComponents_Ready);
259       aRespondToPumpStoppedAlarm_Executing = NOT (aRespondToPumpStoppedAlarm_Ready) AND NOT (
              aRespondToPumpStoppedAlarm_Done);
260       aRespondToPumpStoppedAlarm_Done = (aStep1FixBrokenConnector_Done) AND (
              aStep2DisconnectOldParts_Done) AND (aStep3ConnectNewController_Done) AND (
              aStep4TryLeadBattery_Done) AND (aStep5TryLiIonBattery_Done) AND (
              aStep6TagOldComponents_Done);
261       aStep1FixBrokenConnector_Ready = (aReassembleConnector_Ready) AND (aTryRotatingParts_Ready)
              ;
262       aStep1FixBrokenConnector_Executing = NOT (aStep1FixBrokenConnector_Ready) AND NOT (
              aStep1FixBrokenConnector_Done);
263       aStep1FixBrokenConnector_Done = (aReassembleConnector_Done) AND (aTryRotatingParts_Done);
264       aReassembleConnector_Ready = (hReassembleBrokenConnector_3 = actReady);
265       aReassembleConnector_Executing = NOT (aReassembleConnector_Ready) AND NOT (
              aReassembleConnector_Done);
266       aReassembleConnector_Done = (hReassembleBrokenConnector_3 = actDone);
267       aTryRotatingParts_Ready = (hDisassembleConnector_4 = actReady) AND (hRotateConnectorParts_5
               = actReady) AND (hReassembleBrokenConnector_6 = actReady) AND (NOT
              aTryRotatingParts_Repeating);
268       aTryRotatingParts_Executing = NOT (aTryRotatingParts_Ready) AND NOT (aTryRotatingParts_Done
              );
269       aStep2DisconnectOldParts_Ready = (aDiscPumpFromOldController_Ready) AND (
              aSilenceAlarmOnOldController_Ready) AND (aSetAsideOldComponents_Ready);
270       aStep2DisconnectOldParts_Executing = NOT (aStep2DisconnectOldParts_Ready) AND NOT (
              aStep2DisconnectOldParts_Done);
271       aStep2DisconnectOldParts_Done = (aDiscPumpFromOldController_Done) AND (
              aSilenceAlarmOnOldController_Done) AND (aSetAsideOldComponents_Done);
272       aDiscPumpFromOldController_Ready = (aDiscPumpCableFromAbCable_Ready) AND (
              aDiscPumpCableFromOldController_Ready);
273       aDiscPumpFromOldController_Executing = NOT (aDiscPumpFromOldController_Ready) AND NOT (
              aDiscPumpFromOldController_Done);
274       aDiscPumpFromOldController_Done = (aDiscPumpCableFromAbCable_Done) AND (
              aDiscPumpCableFromOldController_Done);
275       aDiscPumpCableFromAbCable_Ready = (hDiscPumpCableFromAbCable_7 = actReady);
276       aDiscPumpCableFromAbCable_Executing = NOT (aDiscPumpCableFromAbCable_Ready) AND NOT (
              aDiscPumpCableFromAbCable_Done);
277       aDiscPumpCableFromAbCable_Done = (hDiscPumpCableFromAbCable_7 = actDone);
278       aDiscPumpCableFromOldController_Ready = (hDiscPumpCableFromOldController_8 = actReady);
279       aDiscPumpCableFromOldController_Executing = NOT (aDiscPumpCableFromOldController_Ready) AND
```

```
              NOT (aDiscPumpCableFromOldController_Done);
280        aDiscPumpCableFromOldController_Done = (hDiscPumpCableFromOldController_8 = actDone);
281        aSilenceAlarmOnOldController_Ready = (hLoosenOldControllerABCap_9 = actReady);
282        aSilenceAlarmOnOldController_Executing = NOT (aSilenceAlarmOnOldController_Ready) AND NOT (
               aSilenceAlarmOnOldController_Done);
283        aSilenceAlarmOnOldController_Done = (hLoosenOldControllerABCap_9 = actDone);
284        aSetAsideOldComponents_Ready = (hSetAsideOldComponents_10 = actReady);
285        aSetAsideOldComponents_Executing = NOT (aSetAsideOldComponents_Ready) AND NOT (
               aSetAsideOldComponents_Done);
286        aSetAsideOldComponents_Done = (hSetAsideOldComponents_10 = actDone);
287        aStep3ConnectNewController_Ready = (aConPumpCableToNewController_Ready) AND (
               aActivateAlarmOnNewController_Ready);
288        aStep3ConnectNewController_Executing = NOT (aStep3ConnectNewController_Ready) AND NOT (
               aStep3ConnectNewController_Done);
289        aStep3ConnectNewController_Done = (aConPumpCableToNewController_Done) AND (
               aActivateAlarmOnNewController_Done);
290        aConPumpCableToNewController_Ready = (hConPumpCableToNewController_11 = actReady);
291        aConPumpCableToNewController_Executing = NOT (aConPumpCableToNewController_Ready) AND NOT (
               aConPumpCableToNewController_Done);
292        aConPumpCableToNewController_Done = (hConPumpCableToNewController_11 = actDone);
293        aActivateAlarmOnNewController_Ready = (hTightenNewControllerABCap_12 = actReady);
294        aActivateAlarmOnNewController_Executing = NOT (aActivateAlarmOnNewController_Ready) AND NOT
               (aActivateAlarmOnNewController_Done);
295        aActivateAlarmOnNewController_Done = (hTightenNewControllerABCap_12 = actDone);
296        aStep4TryLeadBattery_Ready = (aConLeadBattToNewController_Ready) AND (
               aDiscLeadBattFromNewController_Ready);
297        aStep4TryLeadBattery_Executing = NOT (aStep4TryLeadBattery_Ready) AND NOT (
               aStep4TryLeadBattery_Done);
298        aStep4TryLeadBattery_Done = (aConLeadBattToNewController_Done) AND (
               aDiscLeadBattFromNewController_Done);
299        aConLeadBattToNewController_Ready = (hConLeadBattToNewController_13 = actReady);
300        aConLeadBattToNewController_Executing = NOT (aConLeadBattToNewController_Ready) AND NOT (
               aConLeadBattToNewController_Done);
301        aConLeadBattToNewController_Done = (hConLeadBattToNewController_13 = actDone);
302        aDiscLeadBattFromNewController_Ready = (hDiscLeadBattFromNewController_14 = actReady);
303        aDiscLeadBattFromNewController_Executing = NOT (aDiscLeadBattFromNewController_Ready) AND
               NOT (aDiscLeadBattFromNewController_Done);
304        aDiscLeadBattFromNewController_Done = (hDiscLeadBattFromNewController_14 = actDone);
305        aStep5TryLiIonBattery_Ready = (aCheckLiBatteryLevel_Ready) AND (
               aConnectBattOrCallEmergencyNumber_Ready);
306        aStep5TryLiIonBattery_Executing = NOT (aStep5TryLiIonBattery_Ready) AND NOT (
               aStep5TryLiIonBattery_Done);
307        aStep5TryLiIonBattery_Done = (aCheckLiBatteryLevel_Done) AND (
               aConnectBattOrCallEmergencyNumber_Done);
308        aCheckLiBatteryLevel_Ready = (hDepressBlackButtonOnNewLiBatt_15 = actReady);
309        aCheckLiBatteryLevel_Executing = NOT (aCheckLiBatteryLevel_Ready) AND NOT (
               aCheckLiBatteryLevel_Done);
310        aCheckLiBatteryLevel_Done = (hDepressBlackButtonOnNewLiBatt_15 = actDone);
311        aConnectBattOrCallEmergencyNumber_Ready = (aCallEmergencyNumber_Ready) AND (
               aConnectFullyChargedLiBatt_Ready);
312        aConnectBattOrCallEmergencyNumber_Executing = NOT (aConnectBattOrCallEmergencyNumber_Ready)
                AND NOT (aConnectBattOrCallEmergencyNumber_Done);
313        aConnectBattOrCallEmergencyNumber_Done = (aCallEmergencyNumber_Done) AND (
               aConnectFullyChargedLiBatt_Done);
314        aCallEmergencyNumber_Ready = (hCallEmergencyNumber_16 = actReady);
315        aCallEmergencyNumber_Executing = NOT (aCallEmergencyNumber_Ready) AND NOT (
               aCallEmergencyNumber_Done);
316        aCallEmergencyNumber_Done = (hCallEmergencyNumber_16 = actDone);
317        aConnectFullyChargedLiBatt_Ready = (aConNewLiBattCableToNewController_Ready) AND (
               aConNewLiBattCableToNewLiBatt_Ready) AND (aBreakCircuit_Ready) AND (
               aCallEmergency_Ready);
318        aConnectFullyChargedLiBatt_Executing = NOT (aConnectFullyChargedLiBatt_Ready) AND NOT (
               aConnectFullyChargedLiBatt_Done);
319        aConnectFullyChargedLiBatt_Done = (aConNewLiBattCableToNewController_Done) AND (
               aConNewLiBattCableToNewLiBatt_Done) AND (aBreakCircuit_Done) AND (aCallEmergency_Done)
               ;
320        aConNewLiBattCableToNewController_Ready = (hConNewLiBattCableToNewController_17 = actReady)
               ;
321        aConNewLiBattCableToNewController_Executing = NOT (aConNewLiBattCableToNewController_Ready)
                AND NOT (aConNewLiBattCableToNewController_Done);
322        aConNewLiBattCableToNewController_Done = (hConNewLiBattCableToNewController_17 = actDone);
323        aConNewLiBattCableToNewLiBatt_Ready = (hConNewLiBattCableToNewLiBatt_18 = actReady);
324        aConNewLiBattCableToNewLiBatt_Executing = NOT (aConNewLiBattCableToNewLiBatt_Ready) AND NOT
               (aConNewLiBattCableToNewLiBatt_Done);
325        aConNewLiBattCableToNewLiBatt_Done = (hConNewLiBattCableToNewLiBatt_18 = actDone);
326        aBreakCircuit_Ready = (hDiscNewLiBattCableFromNewController_19 = actReady);
327        aBreakCircuit_Executing = NOT (aBreakCircuit_Ready) AND NOT (aBreakCircuit_Done);
328        aBreakCircuit_Done = (hDiscNewLiBattCableFromNewController_19 = actDone);
329        aCallEmergency_Ready = (hCallEmergencyNumber_20 = actReady);
330        aCallEmergency_Executing = NOT (aCallEmergency_Ready) AND NOT (aCallEmergency_Done);
331        aCallEmergency_Done = (hCallEmergencyNumber_20 = actDone);
```

```
332        aStep6TagOldComponents_Ready = (hRedTagOldComponents_21 = actReady);
333        aStep6TagOldComponents_Executing = NOT (aStep6TagOldComponents_Ready) AND NOT (
               aStep6TagOldComponents_Done);
334        aStep6TagOldComponents_Done = (hRedTagOldComponents_21 = actDone);
335      TRANSITION
336      [
337        (iSpeedSetting = lDesiredSpeed) AND ((aAdjustSpeed_Executing) AND (NOT(
               aIncreaseSetting_Executing) AND NOT(aDecreaseSetting_Executing) AND ((
               aIncreaseSetting_Done) OR (aDecreaseSetting_Done)))) -->
338          aIncreaseSetting_Repeating' = FALSE;
339          aIncreaseSetting_Done' = TRUE;
340          hRotateKnobCounterclockwise_1' = actDone;
341          aDecreaseSetting_Repeating' = FALSE;
342          aDecreaseSetting_Done' = TRUE;
343          hRotateKnobClockwise_2' = actDone;
344        [] aAdjustSpeed_Done -->
345          aIncreaseSetting_Repeating' = FALSE;
346          aIncreaseSetting_Done' = FALSE;
347          hRotateKnobCounterclockwise_1' = actReady;
348          aDecreaseSetting_Repeating' = FALSE;
349          aDecreaseSetting_Done' = FALSE;
350          hRotateKnobClockwise_2' = actReady;
351        []((aIncreaseSetting_Executing) AND (NOT(hRotateKnobCounterclockwise_1 = actExecuting) AND
               ((hRotateKnobCounterclockwise_1 = actDone)))) AND NOT ((iSpeedSetting < lDesiredSpeed)
                AND (NOT(hRotateKnobCounterclockwise_1 = actExecuting) AND ((
               hRotateKnobCounterclockwise_1 = actDone)))) -->
352          aIncreaseSetting_Repeating' = FALSE;
353          aIncreaseSetting_Done' = TRUE;
354          hRotateKnobCounterclockwise_1' = actDone;
355        [](aIncreaseSetting_Executing) AND ((iSpeedSetting < lDesiredSpeed) AND (NOT(
               hRotateKnobCounterclockwise_1 = actExecuting) AND ((hRotateKnobCounterclockwise_1 =
               actDone)))) AND (((aAdjustSpeed_Executing) OR ((aAdjustSpeed_Ready) AND (NOT (
               aRespondToPumpStoppedAlarm_Executing)) AND (iSpeedSetting /= lDesiredSpeed AND iAlarm
               = discreteDevice!NoAlarm) AND NOT (iSpeedSetting = lDesiredSpeed))) AND (
               aDecreaseSetting_Ready)) -->
356          aIncreaseSetting_Repeating' = TRUE;
357          aIncreaseSetting_Done' = FALSE;
358          hRotateKnobCounterclockwise_1' = actReady;
359        []((hRotateKnobCounterclockwise_1 = actReady) AND ((aIncreaseSetting_Executing) OR ((
               aIncreaseSetting_Ready) AND (((aAdjustSpeed_Executing) OR ((aAdjustSpeed_Ready) AND (
               NOT (aRespondToPumpStoppedAlarm_Executing)) AND (iSpeedSetting /= lDesiredSpeed AND
               iAlarm = discreteDevice!NoAlarm) AND NOT (iSpeedSetting = lDesiredSpeed))) AND (
               aDecreaseSetting_Ready)) AND (iSpeedSetting < lDesiredSpeed AND iPage = 8)))) AND
               ready -->
360          hRotateKnobCounterclockwise_1' = actExecuting;
361          hRotateKnobCounterclockwise' = TRUE;
362          submitted' = TRUE;
363        []((aDecreaseSetting_Executing) AND (NOT(hRotateKnobClockwise_2 = actExecuting) AND ((
               hRotateKnobClockwise_2 = actDone)))) AND NOT ((iSpeedSetting > lDesiredSpeed) AND (NOT
               (hRotateKnobClockwise_2 = actExecuting) AND ((hRotateKnobClockwise_2 = actDone)))) -->
364          aDecreaseSetting_Repeating' = FALSE;
365          aDecreaseSetting_Done' = TRUE;
366          hRotateKnobClockwise_2' = actDone;
367        [](aDecreaseSetting_Executing) AND ((iSpeedSetting > lDesiredSpeed) AND (NOT(
               hRotateKnobClockwise_2 = actExecuting) AND ((hRotateKnobClockwise_2 = actDone)))) AND
               (((aAdjustSpeed_Executing) OR ((aAdjustSpeed_Ready) AND (NOT (
               aRespondToPumpStoppedAlarm_Executing)) AND (iSpeedSetting /= lDesiredSpeed AND iAlarm
               = discreteDevice!NoAlarm) AND NOT (iSpeedSetting = lDesiredSpeed))) AND (
               aIncreaseSetting_Ready)) -->
368          aDecreaseSetting_Repeating' = TRUE;
369          aDecreaseSetting_Done' = FALSE;
370          hRotateKnobClockwise_2' = actReady;
371        []((hRotateKnobClockwise_2 = actReady) AND ((aDecreaseSetting_Executing) OR ((
               aDecreaseSetting_Ready) AND (((aAdjustSpeed_Executing) OR ((aAdjustSpeed_Ready) AND (
               NOT (aRespondToPumpStoppedAlarm_Executing)) AND (iSpeedSetting /= lDesiredSpeed AND
               iAlarm = discreteDevice!NoAlarm) AND NOT (iSpeedSetting = lDesiredSpeed))) AND (
               aIncreaseSetting_Ready)) AND (iSpeedSetting > lDesiredSpeed AND iPage = 8)))) AND
               ready -->
372          hRotateKnobClockwise_2' = actExecuting;
373          hRotateKnobClockwise' = TRUE;
374          submitted' = TRUE;
375        [](iAlarm /= discreteDevice!PumpStopped) AND ((aRespondToPumpStoppedAlarm_Executing) AND (
               NOT(aStep1FixBrokenConnector_Executing) AND NOT(aStep2DisconnectOldParts_Executing)
               AND NOT(aStep3ConnectNewController_Executing) AND NOT(aStep4TryLeadBattery_Executing)
               AND NOT(aStep5TryLiIonBattery_Executing) AND NOT(aStep6TagOldComponents_Executing) AND
                ((aStep1FixBrokenConnector_Done) AND (aStep2DisconnectOldParts_Done) AND (
               aStep3ConnectNewController_Done) AND (aStep4TryLeadBattery_Done) AND (
               aStep5TryLiIonBattery_Done) AND (aStep6TagOldComponents_Done)))) -->
376          hReassembleBrokenConnector_3' = actDone;
377          aTryRotatingParts_Repeating' = FALSE;
378          aTryRotatingParts_Done' = TRUE;
```

```
379           hDisassembleConnector_4' = actDone;
380           hRotateConnectorParts_5' = actDone;
381           hReassembleBrokenConnector_6' = actDone;
382           hDiscPumpCableFromAbCable_7' = actDone;
383           hDiscPumpCableFromOldController_8' = actDone;
384           hLoosenOldControllerABCap_9' = actDone;
385           hSetAsideOldComponents_10' = actDone;
386           hConPumpCableToNewController_11' = actDone;
387           hTightenNewControllerABCap_12' = actDone;
388           hConLeadBattToNewController_13' = actDone;
389           hDiscLeadBattFromNewController_14' = actDone;
390           hDepressBlackButtonOnNewLiBatt_15' = actDone;
391           hCallEmergencyNumber_16' = actDone;
392           hConNewLiBattCableToNewController_17' = actDone;
393           hConNewLiBattCableToNewLiBatt_18' = actDone;
394           hDiscNewLiBattCableFromNewController_19' = actDone;
395           hCallEmergencyNumber_20' = actDone;
396           hRedTagOldComponents_21' = actDone;
397         []aRespondToPumpStoppedAlarm_Done -->
398           hReassembleBrokenConnector_3' = actReady;
399           aTryRotatingParts_Repeating' = FALSE;
400           aTryRotatingParts_Done' = FALSE;
401           hDisassembleConnector_4' = actReady;
402           hRotateConnectorParts_5' = actReady;
403           hReassembleBrokenConnector_6' = actReady;
404           hDiscPumpCableFromAbCable_7' = actReady;
405           hDiscPumpCableFromOldController_8' = actReady;
406           hLoosenOldControllerABCap_9' = actReady;
407           hSetAsideOldComponents_10' = actReady;
408           hConPumpCableToNewController_11' = actReady;
409           hTightenNewControllerABCap_12' = actReady;
410           hConLeadBattToNewController_13' = actReady;
411           hDiscLeadBattFromNewController_14' = actReady;
412           hDepressBlackButtonOnNewLiBatt_15' = actReady;
413           hCallEmergencyNumber_16' = actReady;
414           hConNewLiBattCableToNewController_17' = actReady;
415           hConNewLiBattCableToNewLiBatt_18' = actReady;
416           hDiscNewLiBattCableFromNewController_19' = actReady;
417           hCallEmergencyNumber_20' = actReady;
418           hRedTagOldComponents_21' = actReady;
419         [](aStep1FixBrokenConnector_Executing) AND (NOT(aReassembleConnector_Executing) AND NOT(
              aTryRotatingParts_Executing) AND ((aReassembleConnector_Done) AND (
              aTryRotatingParts_Done))) -->
420           hReassembleBrokenConnector_3' = actDone;
421           aTryRotatingParts_Repeating' = FALSE;
422           aTryRotatingParts_Done' = TRUE;
423           hDisassembleConnector_4' = actDone;
424           hRotateConnectorParts_5' = actDone;
425           hReassembleBrokenConnector_6' = actDone;
426         [](iPermanentlyAttachedConnector = Assembled) AND ((aReassembleConnector_Executing) AND (
              NOT(hReassembleBrokenConnector_3 = actExecuting) AND ((hReassembleBrokenConnector_3 =
              actDone)))OR ((aReassembleConnector_Ready) AND (((aStep1FixBrokenConnector_Executing)
              OR ((aStep1FixBrokenConnector_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR
              ((aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
              discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped)))) AND (
              iPermanentlyAttachedConnector = Broken AND iPage = 13)))))) -->
427           hReassembleBrokenConnector_3' = actDone;
428         []((hReassembleBrokenConnector_3 = actReady) AND ((aReassembleConnector_Executing) OR ((
              aReassembleConnector_Ready) AND (((aStep1FixBrokenConnector_Executing) OR ((
              aStep1FixBrokenConnector_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
              aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
              discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped)))) AND (
              iPermanentlyAttachedConnector = Broken AND iPage = 13)))) AND NOT (
              iPermanentlyAttachedConnector = Assembled)))) AND ready -->
429           hReassembleBrokenConnector_3' = actExecuting;
430           hReassembleBrokenConnector' = TRUE;
431           submitted' = TRUE;
432         []((iRotationCounter = 3 AND iPermanentlyAttachedConnector = Assembled) AND ((
              aTryRotatingParts_Executing) AND (NOT(hDisassembleConnector_4 = actExecuting) AND NOT(
              hRotateConnectorParts_5 = actExecuting) AND NOT(hReassembleBrokenConnector_6 =
              actExecuting) AND ((hDisassembleConnector_4 = actDone) AND (hRotateConnectorParts_5 =
              actDone) AND (hReassembleBrokenConnector_6 = actDone)))OR ((aTryRotatingParts_Ready)
              AND (((aStep1FixBrokenConnector_Executing) OR ((aStep1FixBrokenConnector_Ready) AND
              (((aRespondToPumpStoppedAlarm_Executing) OR ((aRespondToPumpStoppedAlarm_Ready) AND (
              NOT (aAdjustSpeed_Executing)) AND (iAlarm = discreteDevice!PumpStopped) AND NOT (
              iAlarm /= discreteDevice!PumpStopped)))) AND (iPermanentlyAttachedConnector = Broken
              AND iPage = 13))) AND (aReassembleConnector_Done))))) AND NOT ((iRotationCounter < 3)
              AND (NOT(hDisassembleConnector_4 = actExecuting) AND NOT(hRotateConnectorParts_5 =
              actExecuting) AND NOT(hReassembleBrokenConnector_6 = actExecuting) AND ((
              hDisassembleConnector_4 = actDone) AND (hRotateConnectorParts_5 = actDone) AND (
              hReassembleBrokenConnector_6 = actDone))) AND NOT (iRotationCounter = 3 AND
```

```
            iPermanentlyAttachedConnector = Assembled)) -->
433         aTryRotatingParts_Repeating' = FALSE;
434         aTryRotatingParts_Done' = TRUE;
435         hDisassembleConnector_4' = actDone;
436         hRotateConnectorParts_5' = actDone;
437         hReassembleBrokenConnector_6' = actDone;
438       [](aTryRotatingParts_Executing) AND ((iRotationCounter < 3) AND (NOT(
            hDisassembleConnector_4 = actExecuting) AND NOT(hRotateConnectorParts_5 = actExecuting
            ) AND NOT(hReassembleBrokenConnector_6 = actExecuting) AND ((hDisassembleConnector_4 =
             actDone) AND (hRotateConnectorParts_5 = actDone) AND (hReassembleBrokenConnector_6 =
            actDone))) AND NOT (iRotationCounter = 3 AND iPermanentlyAttachedConnector = Assembled
            )) AND (((aStep1FixBrokenConnector_Executing) OR ((aStep1FixBrokenConnector_Ready) AND
             (((aRespondToPumpStoppedAlarm_Executing) OR ((aRespondToPumpStoppedAlarm_Ready) AND (
            NOT (aAdjustSpeed_Executing)) AND (iAlarm = discreteDevice!PumpStopped) AND NOT (
            iAlarm /= discreteDevice!PumpStopped)))) AND (iPermanentlyAttachedConnector = Broken
            AND iPage = 13))) AND (aReassembleConnector_Done)) -->
439         aTryRotatingParts_Repeating' = TRUE;
440         aTryRotatingParts_Done' = FALSE;
441         hDisassembleConnector_4' = actReady;
442         hRotateConnectorParts_5' = actReady;
443         hReassembleBrokenConnector_6' = actReady;
444       []((hDisassembleConnector_4 = actReady) AND ((aTryRotatingParts_Executing) OR ((
            aTryRotatingParts_Ready) AND (((aStep1FixBrokenConnector_Executing) OR ((
            aStep1FixBrokenConnector_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
            aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
            discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped)))) AND (
            iPermanentlyAttachedConnector = Broken AND iPage = 13))) AND (
            aReassembleConnector_Done)) AND (iRotationCounter = 0) AND NOT (iRotationCounter = 3
            AND iPermanentlyAttachedConnector = Assembled)))) AND ready -->
445         hDisassembleConnector_4' = actExecuting;
446         hDisassembleConnector' = TRUE;
447         submitted' = TRUE;
448       []((hRotateConnectorParts_5 = actReady) AND ((aTryRotatingParts_Executing) OR ((
            aTryRotatingParts_Ready) AND (((aStep1FixBrokenConnector_Executing) OR ((
            aStep1FixBrokenConnector_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
            aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
            discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped)))) AND (
            iPermanentlyAttachedConnector = Broken AND iPage = 13))) AND (
            aReassembleConnector_Done)) AND (iRotationCounter = 0) AND NOT (iRotationCounter = 3
            AND iPermanentlyAttachedConnector = Assembled))) AND (hDisassembleConnector_4 =
            actDone)) AND ready -->
449         hRotateConnectorParts_5' = actExecuting;
450         hRotateConnectorParts' = TRUE;
451         submitted' = TRUE;
452       []((hReassembleBrokenConnector_6 = actReady) AND ((aTryRotatingParts_Executing) OR ((
            aTryRotatingParts_Ready) AND (((aStep1FixBrokenConnector_Executing) OR ((
            aStep1FixBrokenConnector_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
            aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
            discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped)))) AND (
            iPermanentlyAttachedConnector = Broken AND iPage = 13))) AND (
            aReassembleConnector_Done)) AND (iRotationCounter = 0) AND NOT (iRotationCounter = 3
            AND iPermanentlyAttachedConnector = Assembled))) AND (hRotateConnectorParts_5 =
            actDone)) AND ready -->
453         hReassembleBrokenConnector_6' = actExecuting;
454         hReassembleBrokenConnector' = TRUE;
455         submitted' = TRUE;
456       [](aStep2DisconnectOldParts_Executing) AND (NOT(aDiscPumpFromOldController_Executing) AND
            NOT(aSilenceAlarmOnOldController_Executing) AND NOT(aSetAsideOldComponents_Executing)
            AND ((aDiscPumpFromOldController_Done) AND (aSilenceAlarmOnOldController_Done) AND (
            aSetAsideOldComponents_Done))) -->
457         hDiscPumpCableFromAbCable_7' = actDone;
458         hDiscPumpCableFromOldController_8' = actDone;
459         hLoosenOldControllerABCap_9' = actDone;
460         hSetAsideOldComponents_10' = actDone;
461       [](aDiscPumpFromOldController_Executing) AND (NOT(aDiscPumpCableFromAbCable_Executing) AND
            NOT(aDiscPumpCableFromOldController_Executing) AND ((aDiscPumpCableFromAbCable_Done)
            OR (aDiscPumpCableFromOldController_Done))) -->
462         hDiscPumpCableFromAbCable_7' = actDone;
463         hDiscPumpCableFromOldController_8' = actDone;
464       [](aDiscPumpCableFromAbCable_Executing) AND (NOT(hDiscPumpCableFromAbCable_7 = actExecuting
            ) AND ((hDiscPumpCableFromAbCable_7 = actDone))) -->
465         hDiscPumpCableFromAbCable_7' = actDone;
466       []((hDiscPumpCableFromAbCable_7 = actReady) AND ((aDiscPumpCableFromAbCable_Executing) OR
            ((aDiscPumpCableFromAbCable_Ready) AND (((aDiscPumpFromOldController_Executing) OR ((
            aDiscPumpFromOldController_Ready) AND (((aStep2DisconnectOldParts_Executing) OR ((
            aStep2DisconnectOldParts_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
            aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
            discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
            aStep1FixBrokenConnector_Done)) AND (iPage = 13)))))) AND (
            aDiscPumpCableFromOldController_Ready)) AND (iPumpCableToOldAbCable = Connected AND
            iAbCableToOldController = Connected)))) AND ready -->
```

```
467         hDiscPumpCableFromAbCable_7' = actExecuting;
468         hDiscPumpCableFromAbCable' = TRUE;
469         submitted' = TRUE;
470       [](aDiscPumpCableFromOldController_Executing) AND (NOT(hDiscPumpCableFromOldController_8 =
            actExecuting) AND ((hDiscPumpCableFromOldController_8 = actDone))) -->
471         hDiscPumpCableFromOldController_8' = actDone;
472       []((hDiscPumpCableFromOldController_8 = actReady) AND ((
            aDiscPumpCableFromOldController_Executing) OR ((aDiscPumpCableFromOldController_Ready)
             AND (((aDiscPumpFromOldController_Executing) OR ((aDiscPumpFromOldController_Ready)
            AND (((aStep2DisconnectOldParts_Executing) OR ((aStep2DisconnectOldParts_Ready) AND
            (((aRespondToPumpStoppedAlarm_Executing) OR ((aRespondToPumpStoppedAlarm_Ready) AND (
            NOT (aAdjustSpeed_Executing)) AND (iAlarm = discreteDevice!PumpStopped) AND NOT (
            iAlarm /= discreteDevice!PumpStopped))) AND (aStep1FixBrokenConnector_Done)) AND (
            iPage = 13)))))) AND (aDiscPumpCableFromAbCable_Ready)) AND (iPumpCableToOldController
             = Connected AND iAbCableToOldController = Disconnected)))) AND ready -->
473         hDiscPumpCableFromOldController_8' = actExecuting;
474         hDiscPumpCableFromOldController' = TRUE;
475         submitted' = TRUE;
476       [](iOldControllerABCap = Loosened) AND ((aSilenceAlarmOnOldController_Executing) AND (NOT(
            hLoosenOldControllerABCap_9 = actExecuting) AND ((hLoosenOldControllerABCap_9 =
            actDone)))OR ((aSilenceAlarmOnOldController_Ready) AND (((
            aStep2DisconnectOldParts_Executing) OR ((aStep2DisconnectOldParts_Ready) AND (((
            aRespondToPumpStoppedAlarm_Executing) OR ((aRespondToPumpStoppedAlarm_Ready) AND (NOT
            (aAdjustSpeed_Executing)) AND (iAlarm = discreteDevice!PumpStopped) AND NOT (iAlarm /=
             discreteDevice!PumpStopped))) AND (aStep1FixBrokenConnector_Done)) AND (iPage = 13)))
             AND (aDiscPumpFromOldController_Done)))) -->
477         hLoosenOldControllerABCap_9' = actDone;
478       []((hLoosenOldControllerABCap_9 = actReady) AND ((aSilenceAlarmOnOldController_Executing)
            OR ((aSilenceAlarmOnOldController_Ready) AND (((aStep2DisconnectOldParts_Executing) OR
             ((aStep2DisconnectOldParts_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
            aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
            discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
            aStep1FixBrokenConnector_Done)) AND (iPage = 13))) AND (
            aDiscPumpFromOldController_Done)) AND (iOldControllerABCap = Tightened) AND NOT (
            iOldControllerABCap = Loosened)))) AND ready -->
479         hLoosenOldControllerABCap_9' = actExecuting;
480         hLoosenOldControllerABCap' = TRUE;
481         submitted' = TRUE;
482       [](iOldComponents = SetAside) AND ((aSetAsideOldComponents_Executing) AND (NOT(
            hSetAsideOldComponents_10 = actExecuting) AND ((hSetAsideOldComponents_10 = actDone)))
            OR ((aSetAsideOldComponents_Ready) AND (((aStep2DisconnectOldParts_Executing) OR ((
            aStep2DisconnectOldParts_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
            aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
            discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
            aStep1FixBrokenConnector_Done)) AND (iPage = 13))) AND (
            aSilenceAlarmOnOldController_Done)))) -->
483         hSetAsideOldComponents_10' = actDone;
484       []((hSetAsideOldComponents_10 = actReady) AND ((aSetAsideOldComponents_Executing) OR ((
            aSetAsideOldComponents_Ready) AND (((aStep2DisconnectOldParts_Executing) OR ((
            aStep2DisconnectOldParts_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
            aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
            discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
            aStep1FixBrokenConnector_Done)) AND (iPage = 13))) AND (
            aSilenceAlarmOnOldController_Done)) AND (iOldComponents = AtHand) AND NOT (
            iOldComponents = SetAside)))) AND ready -->
485         hSetAsideOldComponents_10' = actExecuting;
486         hSetAsideOldComponents' = TRUE;
487         submitted' = TRUE;
488       [](iPumpCableToNewController = Connected AND iNewControllerABCap = Tightened) AND ((
            aStep3ConnectNewController_Executing) AND (NOT(aConPumpCableToNewController_Executing)
             AND NOT(aActivateAlarmOnNewController_Executing) AND ((
            aConPumpCableToNewController_Done) AND (aActivateAlarmOnNewController_Done)))OR ((
            aStep3ConnectNewController_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
            aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
            discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
            aStep2DisconnectOldParts_Done)))) -->
489         hConPumpCableToNewController_11' = actDone;
490         hTightenNewControllerABCap_12' = actDone;
491       [](aConPumpCableToNewController_Executing) AND (NOT(hConPumpCableToNewController_11 =
            actExecuting) AND ((hConPumpCableToNewController_11 = actDone))) -->
492         hConPumpCableToNewController_11' = actDone;
493       []((hConPumpCableToNewController_11 = actReady) AND ((
            aConPumpCableToNewController_Executing) OR ((aConPumpCableToNewController_Ready) AND
            (((aStep3ConnectNewController_Executing) OR ((aStep3ConnectNewController_Ready) AND
            (((aRespondToPumpStoppedAlarm_Executing) OR ((aRespondToPumpStoppedAlarm_Ready) AND (
            NOT (aAdjustSpeed_Executing)) AND (iAlarm = discreteDevice!PumpStopped) AND NOT (
            iAlarm /= discreteDevice!PumpStopped))) AND (aStep2DisconnectOldParts_Done)) AND (
            iPumpCableToOldController = Disconnected AND iPumpCableToOldAbCable = Disconnected AND
             iPage = 13) AND NOT (iPumpCableToNewController = Connected AND iNewControllerABCap =
            Tightened))))))) AND ready -->
494         hConPumpCableToNewController_11' = actExecuting;
```

```
495          hConPumpCableToNewController' = TRUE;
496          submitted' = TRUE;
497        [](iNewControllerABCap = Tightened) AND ((aActivateAlarmOnNewController_Executing) AND (NOT
             (hTightenNewControllerABCap_12 = actExecuting) AND ((hTightenNewControllerABCap_12 =
             actDone)))OR ((aActivateAlarmOnNewController_Ready) AND (((
             aStep3ConnectNewController_Executing) OR ((aStep3ConnectNewController_Ready) AND (((
             aRespondToPumpStoppedAlarm_Executing) OR ((aRespondToPumpStoppedAlarm_Ready) AND (NOT
             (aAdjustSpeed_Executing)) AND (iAlarm = discreteDevice!PumpStopped) AND NOT (iAlarm /=
              discreteDevice!PumpStopped))) AND (aStep2DisconnectOldParts_Done)) AND (
             iPumpCableToOldController = Disconnected AND iPumpCableToOldAbCable = Disconnected AND
              iPage = 13) AND NOT (iPumpCableToNewController = Connected AND iNewControllerABCap =
             Tightened))) AND (aConPumpCableToNewController_Done)))) -->
498          hTightenNewControllerABCap_12' = actDone;
499        []((hTightenNewControllerABCap_12 = actReady) AND ((aActivateAlarmOnNewController_Executing
             ) OR ((aActivateAlarmOnNewController_Ready) AND (((
             aStep3ConnectNewController_Executing) OR ((aStep3ConnectNewController_Ready) AND (((
             aRespondToPumpStoppedAlarm_Executing) OR ((aRespondToPumpStoppedAlarm_Ready) AND (NOT
             (aAdjustSpeed_Executing)) AND (iAlarm = discreteDevice!PumpStopped) AND NOT (iAlarm /=
              discreteDevice!PumpStopped))) AND (aStep2DisconnectOldParts_Done)) AND (
             iPumpCableToOldController = Disconnected AND iPumpCableToOldAbCable = Disconnected AND
              iPage = 13) AND NOT (iPumpCableToNewController = Connected AND iNewControllerABCap =
             Tightened))) AND (aConPumpCableToNewController_Done)) AND (iNewControllerABCap =
             Loosened) AND NOT (iNewControllerABCap = Tightened)))) AND ready -->
500          hTightenNewControllerABCap_12' = actExecuting;
501          hTightenNewControllerABCap' = TRUE;
502          submitted' = TRUE;
503        [](aStep4TryLeadBattery_Executing) AND (NOT(aConLeadBattToNewController_Executing) AND NOT(
             aDiscLeadBattFromNewController_Executing) AND ((aConLeadBattToNewController_Done) AND
             (aDiscLeadBattFromNewController_Done))) -->
504          hConLeadBattToNewController_13' = actDone;
505          hDiscLeadBattFromNewController_14' = actDone;
506        [](iLeadBattToNewController = Connected) AND ((aConLeadBattToNewController_Executing) AND (
             NOT(hConLeadBattToNewController_13 = actExecuting) AND ((
             hConLeadBattToNewController_13 = actDone)))OR ((aConLeadBattToNewController_Ready) AND
              (((aStep4TryLeadBattery_Executing) OR ((aStep4TryLeadBattery_Ready) AND (((
             aRespondToPumpStoppedAlarm_Executing) OR ((aRespondToPumpStoppedAlarm_Ready) AND (NOT
             (aAdjustSpeed_Executing)) AND (iAlarm = discreteDevice!PumpStopped) AND NOT (iAlarm /=
              discreteDevice!PumpStopped))) AND (aStep3ConnectNewController_Done)) AND (iPage = 13
             AND iLeadBattToOldController = Disconnected AND
507                             NOT(iLeadBattToYCable = Connected AND iYCableToOldController =
                                    Connected)))))) -->
508          hConLeadBattToNewController_13' = actDone;
509        []((hConLeadBattToNewController_13 = actReady) AND ((aConLeadBattToNewController_Executing)
              OR ((aConLeadBattToNewController_Ready) AND (((aStep4TryLeadBattery_Executing) OR ((
             aStep4TryLeadBattery_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
             aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
             discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
             aStep3ConnectNewController_Done)) AND (iPage = 13 AND iLeadBattToOldController =
             Disconnected AND
510                             NOT(iLeadBattToYCable = Connected AND iYCableToOldController =
                                    Connected))))) AND (iLeadBattToNewController = Disconnected) AND
                                     NOT (iLeadBattToNewController = Connected)))) AND ready -->
511          hConLeadBattToNewController_13' = actExecuting;
512          hConLeadBattToNewController' = TRUE;
513          submitted' = TRUE;
514        [](iLeadBattToNewController = Disconnected) AND ((aDiscLeadBattFromNewController_Executing)
              AND (NOT(hDiscLeadBattFromNewController_14 = actExecuting) AND ((
             hDiscLeadBattFromNewController_14 = actDone)))OR ((
             aDiscLeadBattFromNewController_Ready) AND (((aStep4TryLeadBattery_Executing) OR ((
             aStep4TryLeadBattery_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
             aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
             discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
             aStep3ConnectNewController_Done)) AND (iPage = 13 AND iLeadBattToOldController =
             Disconnected AND
515                             NOT(iLeadBattToYCable = Connected AND iYCableToOldController =
                                    Connected)))) AND (aConLeadBattToNewController_Done)))) -->
516          hDiscLeadBattFromNewController_14' = actDone;
517        []((hDiscLeadBattFromNewController_14 = actReady) AND ((
             aDiscLeadBattFromNewController_Executing) OR ((aDiscLeadBattFromNewController_Ready)
             AND (((aStep4TryLeadBattery_Executing) OR ((aStep4TryLeadBattery_Ready) AND (((
             aRespondToPumpStoppedAlarm_Executing) OR ((aRespondToPumpStoppedAlarm_Ready) AND (NOT
             (aAdjustSpeed_Executing)) AND (iAlarm = discreteDevice!PumpStopped) AND NOT (iAlarm /=
              discreteDevice!PumpStopped))) AND (aStep3ConnectNewController_Done)) AND (iPage = 13
             AND iLeadBattToOldController = Disconnected AND
518                             NOT(iLeadBattToYCable = Connected AND iYCableToOldController =
                                    Connected)))) AND (aConLeadBattToNewController_Done)) AND (
                                    iLeadBattToNewController = Connected) AND NOT (
                                    iLeadBattToNewController = Disconnected)))) AND ready -->
519          hDiscLeadBattFromNewController_14' = actExecuting;
520          hDiscLeadBattFromNewController' = TRUE;
521          submitted' = TRUE;
```

```
522        [](aStep5TryLiIonBattery_Executing) AND (NOT(aCheckLiBatteryLevel_Executing) AND NOT(
             aConnectBattOrCallEmergencyNumber_Executing) AND ((aCheckLiBatteryLevel_Done) AND (
             aConnectBattOrCallEmergencyNumber_Done))) -->
523          hDepressBlackButtonOnNewLiBatt_15' = actDone;
524          hCallEmergencyNumber_16' = actDone;
525          hConNewLiBattCableToNewController_17' = actDone;
526          hConNewLiBattCableToNewLiBatt_18' = actDone;
527          hDiscNewLiBattCableFromNewController_19' = actDone;
528          hCallEmergencyNumber_20' = actDone;
529        [](aCheckLiBatteryLevel_Executing) AND (NOT(hDepressBlackButtonOnNewLiBatt_15 =
             actExecuting) AND ((hDepressBlackButtonOnNewLiBatt_15 = actDone))) -->
530          hDepressBlackButtonOnNewLiBatt_15' = actDone;
531        []((hDepressBlackButtonOnNewLiBatt_15 = actReady) AND ((aCheckLiBatteryLevel_Executing) OR
             ((aCheckLiBatteryLevel_Ready) AND (((aStep5TryLiIonBattery_Executing) OR ((
             aStep5TryLiIonBattery_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
             aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
             discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
             aStep4TryLeadBattery_Done)) AND (iPage = 14))))))) AND ready -->
532          hDepressBlackButtonOnNewLiBatt_15' = actExecuting;
533          hDepressBlackButtonOnNewLiBatt' = TRUE;
534          submitted' = TRUE;
535        [](aConnectBattOrCallEmergencyNumber_Executing) AND (NOT(aCallEmergencyNumber_Executing)
             AND NOT(aConnectFullyChargedLiBatt_Executing) AND ((aCallEmergencyNumber_Done) OR (
             aConnectFullyChargedLiBatt_Done))) -->
536          hCallEmergencyNumber_16' = actDone;
537          hConNewLiBattCableToNewController_17' = actDone;
538          hConNewLiBattCableToNewLiBatt_18' = actDone;
539          hDiscNewLiBattCableFromNewController_19' = actDone;
540          hCallEmergencyNumber_20' = actDone;
541        [](aCallEmergencyNumber_Executing) AND (NOT(hCallEmergencyNumber_16 = actExecuting) AND ((
             hCallEmergencyNumber_16 = actDone))) -->
542          hCallEmergencyNumber_16' = actDone;
543        []((hCallEmergencyNumber_16 = actReady) AND ((aCallEmergencyNumber_Executing) OR ((
             aCallEmergencyNumber_Ready) AND (((aConnectBattOrCallEmergencyNumber_Executing) OR ((
             aConnectBattOrCallEmergencyNumber_Ready) AND (((aStep5TryLiIonBattery_Executing) OR ((
             aStep5TryLiIonBattery_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
             aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
             discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
             aStep4TryLeadBattery_Done)) AND (iPage = 14))) AND (aCheckLiBatteryLevel_Done)))) AND
             (aConnectFullyChargedLiBatt_Ready)) AND (iNewLiBatteryLights < 5)))) AND ready -->
544          hCallEmergencyNumber_16' = actExecuting;
545          hCallEmergencyNumber' = TRUE;
546          submitted' = TRUE;
547        [](aConnectFullyChargedLiBatt_Executing) AND (NOT(
             aConNewLiBattCableToNewController_Executing) AND NOT(
             aConNewLiBattCableToNewLiBatt_Executing) AND NOT(aBreakCircuit_Executing) AND NOT(
             aCallEmergency_Executing) AND ((aConNewLiBattCableToNewController_Done) AND (
             aConNewLiBattCableToNewLiBatt_Done) AND (aBreakCircuit_Done) AND (aCallEmergency_Done)
             )) -->
548          hConNewLiBattCableToNewController_17' = actDone;
549          hConNewLiBattCableToNewLiBatt_18' = actDone;
550          hDiscNewLiBattCableFromNewController_19' = actDone;
551          hCallEmergencyNumber_20' = actDone;
552        [](iNewLiBattCableToNewController = Connected) AND ((
             aConNewLiBattCableToNewController_Executing) AND (NOT(
             hConNewLiBattCableToNewController_17 = actExecuting) AND ((
             hConNewLiBattCableToNewController_17 = actDone)))OR ((
             aConNewLiBattCableToNewController_Ready) AND (((aConnectFullyChargedLiBatt_Executing)
             OR ((aConnectFullyChargedLiBatt_Ready) AND (((
             aConnectBattOrCallEmergencyNumber_Executing) OR ((
             aConnectBattOrCallEmergencyNumber_Ready) AND (((aStep5TryLiIonBattery_Executing) OR ((
             aStep5TryLiIonBattery_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
             aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
             discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
             aStep4TryLeadBattery_Done)) AND (iPage = 14))) AND (aCheckLiBatteryLevel_Done)))) AND
             (aCallEmergencyNumber_Ready)) AND (iNewLiBatteryLights = 5)))))) -->
553          hConNewLiBattCableToNewController_17' = actDone;
554        []((hConNewLiBattCableToNewController_17 = actReady) AND ((
             aConNewLiBattCableToNewController_Executing) OR ((
             aConNewLiBattCableToNewController_Ready) AND (((aConnectFullyChargedLiBatt_Executing)
             OR ((aConnectFullyChargedLiBatt_Ready) AND (((
             aConnectBattOrCallEmergencyNumber_Executing) OR ((
             aConnectBattOrCallEmergencyNumber_Ready) AND (((aStep5TryLiIonBattery_Executing) OR ((
             aStep5TryLiIonBattery_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
             aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
             discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
             aStep4TryLeadBattery_Done)) AND (iPage = 14))) AND (aCheckLiBatteryLevel_Done)))) AND
             (aCallEmergencyNumber_Ready)) AND (iNewLiBatteryLights = 5)))) AND (
             iNewLiBattCableToNewController = Disconnected) AND NOT (iNewLiBattCableToNewController
              = Connected)))) AND ready -->
555          hConNewLiBattCableToNewController_17' = actExecuting;
```

```
556            hConNewLiBattCableToNewController' = TRUE;
557            submitted' = TRUE;
558          [](iNewLiBattCableToNewLiBatt = Connected) AND ((aConNewLiBattCableToNewLiBatt_Executing)
               AND (NOT(hConNewLiBattCableToNewLiBatt_18 = actExecuting) AND ((
               hConNewLiBattCableToNewLiBatt_18 = actDone)))OR ((aConNewLiBattCableToNewLiBatt_Ready)
                AND (((aConnectFullyChargedLiBatt_Executing) OR ((aConnectFullyChargedLiBatt_Ready)
               AND (((aConnectBattOrCallEmergencyNumber_Executing) OR ((
               aConnectBattOrCallEmergencyNumber_Ready) AND (((aStep5TryLiIonBattery_Executing) OR ((
               aStep5TryLiIonBattery_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
               aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
               discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
               aStep4TryLeadBattery_Done)) AND (iPage = 14))) AND (aCheckLiBatteryLevel_Done)))) AND
               (aCallEmergencyNumber_Ready)) AND (iNewLiBatteryLights = 5))) AND (
               aConNewLiBattCableToNewController_Done)))) -->
559            hConNewLiBattCableToNewLiBatt_18' = actDone;
560          []((hConNewLiBattCableToNewLiBatt_18 = actReady) AND ((
               aConNewLiBattCableToNewLiBatt_Executing) OR ((aConNewLiBattCableToNewLiBatt_Ready) AND
                (((aConnectFullyChargedLiBatt_Executing) OR ((aConnectFullyChargedLiBatt_Ready) AND
               (((aConnectBattOrCallEmergencyNumber_Executing) OR ((
               aConnectBattOrCallEmergencyNumber_Ready) AND (((aStep5TryLiIonBattery_Executing) OR ((
               aStep5TryLiIonBattery_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
               aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
               discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
               aStep4TryLeadBattery_Done)) AND (iPage = 14))) AND (aCheckLiBatteryLevel_Done)))) AND
               (aCallEmergencyNumber_Ready)) AND (iNewLiBatteryLights = 5))) AND (
               aConNewLiBattCableToNewController_Done)) AND (iNewLiBattCableToNewLiBatt =
               Disconnected) AND NOT (iNewLiBattCableToNewLiBatt = Connected)))) AND ready -->
561            hConNewLiBattCableToNewLiBatt_18' = actExecuting;
562            hConNewLiBattCableToNewLiBatt' = TRUE;
563            submitted' = TRUE;
564          [](iPowerLight = 0) AND ((aBreakCircuit_Executing) AND (NOT(
               hDiscNewLiBattCableFromNewController_19 = actExecuting) AND ((
               hDiscNewLiBattCableFromNewController_19 = actDone)))OR ((aBreakCircuit_Ready) AND (((
               aConnectFullyChargedLiBatt_Executing) OR ((aConnectFullyChargedLiBatt_Ready) AND (((
               aConnectBattOrCallEmergencyNumber_Executing) OR ((
               aConnectBattOrCallEmergencyNumber_Ready) AND (((aStep5TryLiIonBattery_Executing) OR ((
               aStep5TryLiIonBattery_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
               aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
               discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
               aStep4TryLeadBattery_Done)) AND (iPage = 14))) AND (aCheckLiBatteryLevel_Done)))) AND
               (aCallEmergencyNumber_Ready)) AND (iNewLiBatteryLights = 5))) AND (
               aConNewLiBattCableToNewLiBatt_Done)))) -->
565            hDiscNewLiBattCableFromNewController_19' = actDone;
566          []((hDiscNewLiBattCableFromNewController_19 = actReady) AND ((aBreakCircuit_Executing) OR
               ((aBreakCircuit_Ready) AND (((aConnectFullyChargedLiBatt_Executing) OR ((
               aConnectFullyChargedLiBatt_Ready) AND (((aConnectBattOrCallEmergencyNumber_Executing)
               OR ((aConnectBattOrCallEmergencyNumber_Ready) AND (((aStep5TryLiIonBattery_Executing)
               OR ((aStep5TryLiIonBattery_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
               aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
               discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
               aStep4TryLeadBattery_Done)) AND (iPage = 14))) AND (aCheckLiBatteryLevel_Done)))) AND
               (aCallEmergencyNumber_Ready)) AND (iNewLiBatteryLights = 5))) AND (
               aConNewLiBattCableToNewLiBatt_Done)) AND NOT (iPowerLight = 0)))) AND ready -->
567            hDiscNewLiBattCableFromNewController_19' = actExecuting;
568            hDiscNewLiBattCableFromNewController' = TRUE;
569            submitted' = TRUE;
570          [](aCallEmergency_Executing) AND (NOT(hCallEmergencyNumber_20 = actExecuting) AND ((
               hCallEmergencyNumber_20 = actDone))) -->
571            hCallEmergencyNumber_20' = actDone;
572          []((hCallEmergencyNumber_20 = actReady) AND ((aCallEmergency_Executing) OR ((
               aCallEmergency_Ready) AND (((aConnectFullyChargedLiBatt_Executing) OR ((
               aConnectFullyChargedLiBatt_Ready) AND (((aConnectBattOrCallEmergencyNumber_Executing)
               OR ((aConnectBattOrCallEmergencyNumber_Ready) AND (((aStep5TryLiIonBattery_Executing)
               OR ((aStep5TryLiIonBattery_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
               aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
               discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
               aStep4TryLeadBattery_Done)) AND (iPage = 14))) AND (aCheckLiBatteryLevel_Done)))) AND
               (aCallEmergencyNumber_Ready)) AND (iNewLiBatteryLights = 5))) AND (aBreakCircuit_Done)
               )))) AND ready -->
573            hCallEmergencyNumber_20' = actExecuting;
574            hCallEmergencyNumber' = TRUE;
575            submitted' = TRUE;
576          [](iOldComponentTags = redTagged) AND ((aStep6TagOldComponents_Executing) AND (NOT(
               hRedTagOldComponents_21 = actExecuting) AND ((hRedTagOldComponents_21 = actDone)))OR
               ((aStep6TagOldComponents_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
               aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
               discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
               aStep5TryLiIonBattery_Done)))) -->
577            hRedTagOldComponents_21' = actDone;
578          []((hRedTagOldComponents_21 = actReady) AND ((aStep6TagOldComponents_Executing) OR ((
               aStep6TagOldComponents_Ready) AND (((aRespondToPumpStoppedAlarm_Executing) OR ((
```

```
            aRespondToPumpStoppedAlarm_Ready) AND (NOT (aAdjustSpeed_Executing)) AND (iAlarm =
            discreteDevice!PumpStopped) AND NOT (iAlarm /= discreteDevice!PumpStopped))) AND (
            aStep5TryLiIonBattery_Done)) AND (iPage = 14 AND iOldComponentTags = notRedTagged) AND
             NOT (iOldComponentTags = redTagged)))) AND ready -->
579         hRedTagOldComponents_21' = actExecuting;
580         hRedTagOldComponents' = TRUE;
581         submitted' = TRUE;
582       []submitted AND NOT ready -->
583         submitted' = FALSE;
584         hRotateKnobCounterclockwise_1' = IF hRotateKnobCounterclockwise_1 = actExecuting THEN
                 actDone ELSE hRotateKnobCounterclockwise_1 ENDIF;
585         hRotateKnobClockwise_2' = IF hRotateKnobClockwise_2 = actExecuting THEN actDone ELSE
                 hRotateKnobClockwise_2 ENDIF;
586         hReassembleBrokenConnector_3' = IF hReassembleBrokenConnector_3 = actExecuting THEN
                 actDone ELSE hReassembleBrokenConnector_3 ENDIF;
587         hDisassembleConnector_4' = IF hDisassembleConnector_4 = actExecuting THEN actDone ELSE
                 hDisassembleConnector_4 ENDIF;
588         hRotateConnectorParts_5' = IF hRotateConnectorParts_5 = actExecuting THEN actDone ELSE
                 hRotateConnectorParts_5 ENDIF;
589         hReassembleBrokenConnector_6' = IF hReassembleBrokenConnector_6 = actExecuting THEN
                 actDone ELSE hReassembleBrokenConnector_6 ENDIF;
590         hDiscPumpCableFromAbCable_7' = IF hDiscPumpCableFromAbCable_7 = actExecuting THEN actDone
                  ELSE hDiscPumpCableFromAbCable_7 ENDIF;
591         hDiscPumpCableFromOldController_8' = IF hDiscPumpCableFromOldController_8 = actExecuting
                 THEN actDone ELSE hDiscPumpCableFromOldController_8 ENDIF;
592         hLoosenOldControllerABCap_9' = IF hLoosenOldControllerABCap_9 = actExecuting THEN actDone
                  ELSE hLoosenOldControllerABCap_9 ENDIF;
593         hSetAsideOldComponents_10' = IF hSetAsideOldComponents_10 = actExecuting THEN actDone
                 ELSE hSetAsideOldComponents_10 ENDIF;
594         hConPumpCableToNewController_11' = IF hConPumpCableToNewController_11 = actExecuting THEN
                  actDone ELSE hConPumpCableToNewController_11 ENDIF;
595         hTightenNewControllerABCap_12' = IF hTightenNewControllerABCap_12 = actExecuting THEN
                 actDone ELSE hTightenNewControllerABCap_12 ENDIF;
596         hConLeadBattToNewController_13' = IF hConLeadBattToNewController_13 = actExecuting THEN
                 actDone ELSE hConLeadBattToNewController_13 ENDIF;
597         hDiscLeadBattFromNewController_14' = IF hDiscLeadBattFromNewController_14 = actExecuting
                 THEN actDone ELSE hDiscLeadBattFromNewController_14 ENDIF;
598         hDepressBlackButtonOnNewLiBatt_15' = IF hDepressBlackButtonOnNewLiBatt_15 = actExecuting
                 THEN actDone ELSE hDepressBlackButtonOnNewLiBatt_15 ENDIF;
599         hCallEmergencyNumber_16' = IF hCallEmergencyNumber_16 = actExecuting THEN actDone ELSE
                 hCallEmergencyNumber_16 ENDIF;
600         hConNewLiBattCableToNewController_17' = IF hConNewLiBattCableToNewController_17 =
                 actExecuting THEN actDone ELSE hConNewLiBattCableToNewController_17 ENDIF;
601         hConNewLiBattCableToNewLiBatt_18' = IF hConNewLiBattCableToNewLiBatt_18 = actExecuting
                 THEN actDone ELSE hConNewLiBattCableToNewLiBatt_18 ENDIF;
602         hDiscNewLiBattCableFromNewController_19' = IF hDiscNewLiBattCableFromNewController_19 =
                 actExecuting THEN actDone ELSE hDiscNewLiBattCableFromNewController_19 ENDIF;
603         hCallEmergencyNumber_20' = IF hCallEmergencyNumber_20 = actExecuting THEN actDone ELSE
                 hCallEmergencyNumber_20 ENDIF;
604         hRedTagOldComponents_21' = IF hRedTagOldComponents_21 = actExecuting THEN actDone ELSE
                 hRedTagOldComponents_21 ENDIF;
605         hCallEmergencyNumber' = FALSE;
606         hRedTagOldComponents' = FALSE;
607         hSetAsideOldComponents' = FALSE;
608         hRotateConnectorParts' = FALSE;
609         hDisassembleConnector' = FALSE;
610         hReassembleBrokenConnector' = FALSE;
611         hDiscPumpCableFromAbCable' = FALSE;
612         hDiscPumpCableFromOldController' = FALSE;
613         hDiscLeadBattFromNewController' = FALSE;
614         hDiscNewLiBattCableFromNewController' = FALSE;
615         hConNewLiBattCableToNewController' = FALSE;
616         hConNewLiBattCableToNewLiBatt' = FALSE;
617         hDiscNewLiBattCableFromNewLiBatt' = FALSE;
618         hConLeadBattToNewController' = FALSE;
619         hConPumpCableToNewController' = FALSE;
620         hConNewLeadBattToNewController' = FALSE;
621         hDepressBlackButtonOnNewLiBatt' = FALSE;
622         hRotateKnobClockwise' = FALSE;
623         hRotateKnobCounterclockwise' = FALSE;
624         hTightenNewControllerABCap' = FALSE;
625         hLoosenOldControllerABCap' = FALSE;
626         hLoosenNewControllerABCap' = FALSE;
627     ];
628   END;
629 END
```

H.2.3 Affordance Model

```
1 cavemen: CONTEXT =
```

```
2  BEGIN
3 position: TYPE = {up, down, back, forth};
4 translate: TYPE = {left, right};
5 orient: TYPE = {pitch_back, pitch_forth, yaw_left, yaw_right, roll_left, roll_right};
6 abilities: TYPE = [#positionable: ARRAY position OF BOOLEAN, translatable: ARRAY translate OF
     BOOLEAN, orientable: ARRAY orient OF BOOLEAN#];
7 topological: TYPE = {disjoint_to, touching, covering, overlapping};
8 directional: TYPE = {left_of, right_of, top_of, bottom_of, front_of, back_of};
9 relations: TYPE = ARRAY directional OF topological;
10 aoConnectorPart1_rels: TYPE = [#aoConnectorPart2: relations#];
11 aoConnectorPart2_rels: TYPE = [#aoConnectorPart1: relations#];
12 aoNCBatteryInput_rels: TYPE = [#aoNBCControllerOutput: relations, aoLeadBattControllerOutput:
     relations, aoOBCControllerOutput: relations, aoYCControllerOutput: relations#];
13 aoOCBatteryInput_rels: TYPE = [#aoLeadBattControllerOutput: relations, aoNBCControllerOutput:
     relations, aoOBCControllerOutput: relations#];
14 aoYCableBatteryInput1_rels: TYPE = [#aoLeadBattControllerOutput: relations, aoNBCControllerOutput
     : relations, aoOBCControllerOutput: relations#];
15 aoYCableBatteryInput2_rels: TYPE = [#aoLeadBattControllerOutput: relations, aoNBCControllerOutput
     : relations, aoOBCControllerOutput: relations#];
16 aoOBBattCableInput_rels: TYPE = [#aoNBCBatteryOutput: relations, aoOBCBatteryOutput: relations#];
17 aoNBBattCableInput_rels: TYPE = [#aoNBCBatteryOutput: relations, aoOBCBatteryOutput: relations#];
18 aoOCPumpInput_rels: TYPE = [#aoPCControllerOutput: relations#];
19 aoNCPumpInput_rels: TYPE = [#aoPCControllerOutput: relations#];
20 aoACPumpInput_rels: TYPE = [#aoPCControllerOutput: relations#];
21 affords(feature: BOOLEAN, ability: BOOLEAN): BOOLEAN = IF feature AND ability THEN TRUE ELSE
     FALSE ENDIF;
22 affordance: MODULE =
23  BEGIN
24   INPUT aoConnectorPart1: aoConnectorPart1_rels
25   INPUT aoConnectorPart2: aoConnectorPart2_rels
26   INPUT aoNCBatteryInput: aoNCBatteryInput_rels
27   INPUT aoOCBatteryInput: aoOCBatteryInput_rels
28   INPUT aoYCableBatteryInput1: aoYCableBatteryInput1_rels
29   INPUT aoYCableBatteryInput2: aoYCableBatteryInput2_rels
30   INPUT aoOBBattCableInput: aoOBBattCableInput_rels
31   INPUT aoNBBattCableInput: aoNBBattCableInput_rels
32   INPUT aoOCPumpInput: aoOCPumpInput_rels
33   INPUT aoNCPumpInput: aoNCPumpInput_rels
34   INPUT aoACPumpInput: aoACPumpInput_rels
35   INPUT pPumpOperator_aoConnectorPart1: abilities
36   INPUT pPumpOperator_aoConnectorPart2: abilities
37   INPUT pPumpOperator_mNewController: abilities
38   INPUT pPumpOperator_aoNBCControllerOutput: abilities
39   INPUT pPumpOperator_aoLeadBattControllerOutput: abilities
40   INPUT pPumpOperator_mOldLiIonBattery: abilities
41   INPUT pPumpOperator_aoNBCBatteryOutput: abilities
42   INPUT pPumpOperator_mNewLiIonBattery: abilities
43   INPUT pPumpOperator_aoOBCControllerOutput: abilities
44   INPUT pPumpOperator_aoPCControllerOutput: abilities
45   INPUT pPumpOperator_mOldController: abilities
46   INPUT pPumpOperator_aoACPumpInput: abilities
47   LOCAL ability_RotatePart1: BOOLEAN
48   LOCAL ability_RotatePart2: BOOLEAN
49   LOCAL ability_AssemblePart1: BOOLEAN
50   LOCAL ability_AssemblePart2: BOOLEAN
51   LOCAL ability_DisassemblePart1: BOOLEAN
52   LOCAL ability_DisassemblePart2: BOOLEAN
53   LOCAL ability_MoveNewControllerBack: BOOLEAN
54   LOCAL ability_MoveNewLiBattCableControllerOutputBack: BOOLEAN
55   LOCAL ability_MoveLeadBattCableControllerOutputBack: BOOLEAN
56   LOCAL ability_MoveNewController: BOOLEAN
57   LOCAL ability_MoveLeadBattControllerOutput: BOOLEAN
58   LOCAL ability_MoveOldLiBattery: BOOLEAN
59   LOCAL ability_MoveNewBattCableBatteryOutput: BOOLEAN
60   LOCAL ability_MoveNewLiBattery: BOOLEAN
61   LOCAL ability_MoveNewBattCableControllerOutput: BOOLEAN
62   LOCAL ability_MoveOldBattCableControllerOutput: BOOLEAN
63   LOCAL ability_MovePumpCableOutput: BOOLEAN
64   LOCAL ability_MoveOldControllerBack: BOOLEAN
65   LOCAL ability_MovePumpCableControllerOutputBack: BOOLEAN
66   LOCAL ability_MoveAbCablePumpInputBack: BOOLEAN
67   LOCAL feature_ConnectorPartsRotatable: BOOLEAN
68   LOCAL feature_ConnectorPartsAssemblable: BOOLEAN
69   LOCAL feature_ConnectorPartsDisassemblable: BOOLEAN
70   LOCAL feature_NewLiBattCableDisconnectableFromNewController: BOOLEAN
71   LOCAL feature_LeadBattDisconnectableFromNewController: BOOLEAN
72   LOCAL feature_LeadBattConnectableToNewController: BOOLEAN
73   LOCAL feature_NewLiBattCableConnectableToOldLiBatt: BOOLEAN
74   LOCAL feature_NewLiBattCableConnectableToNewLiBatt: BOOLEAN
75   LOCAL feature_NewLiBattCableConnectableToNewController: BOOLEAN
76   LOCAL feature_OldLiBattCableConnectableToNewController: BOOLEAN
```

```
77    LOCAL feature_PumpCableConnectableToNewController: BOOLEAN
78    LOCAL feature_PumpCableDisconnectableFromOldController: BOOLEAN
79    LOCAL feature_PumpCableDisconnectableFromAbCable: BOOLEAN
80    OUTPUT ConnectorPartsRotatable: BOOLEAN
81    OUTPUT ConnectorPartsAssemblable: BOOLEAN
82    OUTPUT ConnectorPartsDisassemblable: BOOLEAN
83    OUTPUT NewLiBattCableDisconnectableFromNewController: BOOLEAN
84    OUTPUT LeadBattDisconnectableFromNewController: BOOLEAN
85    OUTPUT LeadBattConnectableToNewController: BOOLEAN
86    OUTPUT NewLiBattCableConnectableToOldLiBatt: BOOLEAN
87    OUTPUT NewLiBattCableConnectableToNewLiBatt: BOOLEAN
88    OUTPUT NewLiBattCableConnectableToNewController: BOOLEAN
89    OUTPUT OldLiBattCableConnectableToNewController: BOOLEAN
90    OUTPUT PumpCableConnectableToNewController: BOOLEAN
91    OUTPUT PumpCableDisconnectableFromOldController: BOOLEAN
92    OUTPUT PumpCableDisconnectableFromAbCable: BOOLEAN
93     DEFINITION
94      ability_RotatePart1 =
95    pPumpOperator_aoConnectorPart1.orientable[roll_right] = TRUE AND
96    pPumpOperator_aoConnectorPart1.orientable[roll_left] = TRUE;
97      ability_RotatePart2 =
98    pPumpOperator_aoConnectorPart2.orientable[roll_right] = TRUE AND
99    pPumpOperator_aoConnectorPart2.orientable[roll_left] = TRUE;
100     feature_ConnectorPartsRotatable =
101       aoConnectorPart1.aoConnectorPart2[front_of] = disjoint_to AND
102       aoConnectorPart2.aoConnectorPart1[back_of] = disjoint_to;
103 ConnectorPartsRotatable = affords(feature_ConnectorPartsRotatable, ability_RotatePart1 AND
        ability_RotatePart2);
104     ability_AssemblePart1 =
105   pPumpOperator_aoConnectorPart1.positionable[back] = TRUE;
106     ability_AssemblePart2 =
107   pPumpOperator_aoConnectorPart2.positionable[forth] = TRUE;
108     feature_ConnectorPartsAssemblable =
109       aoConnectorPart1.aoConnectorPart2[front_of] /= covering;
110 ConnectorPartsAssemblable = affords(feature_ConnectorPartsAssemblable, ability_AssemblePart1 AND
        ability_AssemblePart2);
111     ability_DisassemblePart1 =
112   pPumpOperator_aoConnectorPart1.positionable[forth] = TRUE;
113     ability_DisassemblePart2 =
114   pPumpOperator_aoConnectorPart2.positionable[back] = TRUE;
115     feature_ConnectorPartsDisassemblable =
116       aoConnectorPart1.aoConnectorPart2[front_of] = covering;
117 ConnectorPartsDisassemblable = affords(feature_ConnectorPartsDisassemblable,
        ability_DisassemblePart1 AND ability_DisassemblePart2);
118     ability_MoveNewControllerBack =
119   pPumpOperator_mNewController.positionable[back] = TRUE;
120     ability_MoveNewLiBattCableControllerOutputBack =
121   pPumpOperator_aoNBCControllerOutput.positionable[back] = TRUE;
122     feature_NewLiBattCableDisconnectableFromNewController =
123       FORALL(x: directional): aoNCBatteryInput.aoNBCControllerOutput[x] = covering;
124 NewLiBattCableDisconnectableFromNewController = affords(
        feature_NewLiBattCableDisconnectableFromNewController, ability_MoveNewControllerBack AND
        ability_MoveNewLiBattCableControllerOutputBack);
125     ability_MoveLeadBattCableControllerOutputBack =
126   pPumpOperator_aoLeadBattControllerOutput.positionable[back] = TRUE;
127     feature_LeadBattDisconnectableFromNewController =
128       FORALL(x: directional): aoNCBatteryInput.aoLeadBattControllerOutput[x] = covering;
129 LeadBattDisconnectableFromNewController = affords(feature_LeadBattDisconnectableFromNewController
        , ability_MoveNewControllerBack AND ability_MoveLeadBattCableControllerOutputBack);
130     ability_MoveNewController =
131   FORALL(x: orient): pPumpOperator_mNewController.orientable[x] = TRUE AND
132   FORALL(x: translate): pPumpOperator_mNewController.translatable[x] = TRUE AND
133   FORALL(x: position): pPumpOperator_mNewController.positionable[x] = TRUE;
134     ability_MoveLeadBattControllerOutput =
135   pPumpOperator_aoLeadBattControllerOutput.orientable[pitch_back] = TRUE AND
136   pPumpOperator_aoLeadBattControllerOutput.orientable[pitch_forth] = TRUE AND
137   pPumpOperator_aoLeadBattControllerOutput.orientable[yaw_left] = TRUE AND
138   pPumpOperator_aoLeadBattControllerOutput.orientable[yaw_right] = TRUE AND
139   FORALL(x: translate): pPumpOperator_aoLeadBattControllerOutput.translatable[x] = TRUE AND
140   FORALL(x: position): pPumpOperator_aoLeadBattControllerOutput.positionable[x] = TRUE;
141     feature_LeadBattConnectableToNewController =
142       FORALL(x: directional): aoNCBatteryInput.aoLeadBattControllerOutput[x] /= covering AND
143       FORALL(x: directional): aoNCBatteryInput.aoOBCControllerOutput[x] /= covering AND
144       FORALL(x: directional): aoNCBatteryInput.aoNBCControllerOutput[x] /= covering AND
145       FORALL(x: directional): aoNCBatteryInput.aoYCControllerOutput[x] /= covering AND
146       FORALL(x: directional): aoOCBatteryInput.aoLeadBattControllerOutput[x] /= covering AND
147       FORALL(x: directional): aoYCableBatteryInput1.aoLeadBattControllerOutput[x] /= covering AND
148       FORALL(x: directional): aoYCableBatteryInput2.aoLeadBattControllerOutput[x] /= covering;
149 LeadBattConnectableToNewController = affords(feature_LeadBattConnectableToNewController,
        ability_MoveNewController AND ability_MoveLeadBattControllerOutput);
150     ability_MoveOldLiBattery =
```

```
151      pPumpOperator_mOldLiIonBattery.orientable[pitch_back] = TRUE AND
152      pPumpOperator_mOldLiIonBattery.orientable[pitch_forth] = TRUE AND
153      pPumpOperator_mOldLiIonBattery.orientable[yaw_left] = TRUE AND
154      pPumpOperator_mOldLiIonBattery.orientable[yaw_right] = TRUE AND
155      FORALL(x: translate): pPumpOperator_mOldLiIonBattery.translatable[x] = TRUE AND
156      FORALL(x: position): pPumpOperator_mOldLiIonBattery.positionable[x] = TRUE;
157        ability_MoveNewBattCableBatteryOutput =
158      pPumpOperator_aoNBCBatteryOutput.orientable[pitch_back] = TRUE AND
159      pPumpOperator_aoNBCBatteryOutput.orientable[pitch_forth] = TRUE AND
160      pPumpOperator_aoNBCBatteryOutput.orientable[yaw_left] = TRUE AND
161      pPumpOperator_aoNBCBatteryOutput.orientable[yaw_right] = TRUE AND
162      FORALL(x: translate): pPumpOperator_aoNBCBatteryOutput.translatable[x] = TRUE AND
163      FORALL(x: position): pPumpOperator_aoNBCBatteryOutput.positionable[x] = TRUE;
164        feature_NewLiBattCableConnectableToOldLiBatt =
165          FORALL(x: directional): aoOBBattCableInput.aoNBCBatteryOutput[x] /= covering AND
166          FORALL(x: directional): aoOBBattCableInput.aoOBCBatteryOutput[x] /= covering AND
167          FORALL(x: directional): aoNBBattCableInput.aoNBCBatteryOutput[x] /= covering;
168    NewLiBattCableConnectableToOldLiBatt = affords(feature_NewLiBattCableConnectableToOldLiBatt,
           ability_MoveOldLiBattery AND ability_MoveNewBattCableBatteryOutput);
169        ability_MoveNewLiBattery =
170      pPumpOperator_mNewLiIonBattery.orientable[pitch_back] = TRUE AND
171      pPumpOperator_mNewLiIonBattery.orientable[pitch_forth] = TRUE AND
172      pPumpOperator_mNewLiIonBattery.orientable[yaw_left] = TRUE AND
173      pPumpOperator_mNewLiIonBattery.orientable[yaw_right] = TRUE AND
174      FORALL(x: translate): pPumpOperator_mNewLiIonBattery.translatable[x] = TRUE AND
175      FORALL(x: position): pPumpOperator_mNewLiIonBattery.positionable[x] = TRUE;
176        feature_NewLiBattCableConnectableToNewLiBatt =
177          FORALL(x: directional): aoOBBattCableInput.aoNBCBatteryOutput[x] /= covering AND
178          FORALL(x: directional): aoNBBattCableInput.aoNBCBatteryOutput[x] /= covering AND
179          FORALL(x: directional): aoNBBattCableInput.aoOBCBatteryOutput[x] /= covering;
180    NewLiBattCableConnectableToNewLiBatt = affords(feature_NewLiBattCableConnectableToNewLiBatt,
           ability_MoveNewLiBattery AND ability_MoveNewBattCableBatteryOutput);
181        ability_MoveNewBattCableControllerOutput =
182      pPumpOperator_aoNBCControllerOutput.orientable[pitch_back] = TRUE AND
183      pPumpOperator_aoNBCControllerOutput.orientable[pitch_forth] = TRUE AND
184      pPumpOperator_aoNBCControllerOutput.orientable[yaw_left] = TRUE AND
185      pPumpOperator_aoNBCControllerOutput.orientable[yaw_right] = TRUE AND
186      FORALL(x: translate): pPumpOperator_aoNBCControllerOutput.translatable[x] = TRUE AND
187      FORALL(x: position): pPumpOperator_aoNBCControllerOutput.positionable[x] = TRUE;
188        feature_NewLiBattCableConnectableToNewController =
189          FORALL(x: directional): aoNCBatteryInput.aoOBCControllerOutput[x] /= covering AND
190          FORALL(x: directional): aoNCBatteryInput.aoNBCControllerOutput[x] /= covering AND
191          FORALL(x: directional): aoNCBatteryInput.aoYCControllerOutput[x] /= covering AND
192          FORALL(x: directional): aoNCBatteryInput.aoLeadBattControllerOutput[x] /= covering AND
193          FORALL(x: directional): aoOCBatteryInput.aoNBCControllerOutput[x] /= covering AND
194          FORALL(x: directional): aoYCableBatteryInput1.aoNBCControllerOutput[x] /= covering AND
195          FORALL(x: directional): aoYCableBatteryInput2.aoNBCControllerOutput[x] /= covering;
196    NewLiBattCableConnectableToNewController = affords(
           feature_NewLiBattCableConnectableToNewController, ability_MoveNewController AND
           ability_MoveNewBattCableControllerOutput);
197        ability_MoveOldBattCableControllerOutput =
198      pPumpOperator_aoOBCControllerOutput.orientable[pitch_back] = TRUE AND
199      pPumpOperator_aoOBCControllerOutput.orientable[pitch_forth] = TRUE AND
200      pPumpOperator_aoOBCControllerOutput.orientable[yaw_left] = TRUE AND
201      pPumpOperator_aoOBCControllerOutput.orientable[yaw_right] = TRUE AND
202      FORALL(x: translate): pPumpOperator_aoOBCControllerOutput.translatable[x] = TRUE AND
203      FORALL(x: position): pPumpOperator_aoOBCControllerOutput.positionable[x] = TRUE;
204        feature_OldLiBattCableConnectableToNewController =
205          FORALL(x: directional): aoNCBatteryInput.aoOBCControllerOutput[x] /= covering AND
206          FORALL(x: directional): aoNCBatteryInput.aoNBCControllerOutput[x] /= covering AND
207          FORALL(x: directional): aoNCBatteryInput.aoYCControllerOutput[x] /= covering AND
208          FORALL(x: directional): aoNCBatteryInput.aoLeadBattControllerOutput[x] /= covering AND
209          FORALL(x: directional): aoYCableBatteryInput1.aoOBCControllerOutput[x] /= covering AND
210          FORALL(x: directional): aoYCableBatteryInput2.aoOBCControllerOutput[x] /= covering AND
211          FORALL(x: directional): aoOCBatteryInput.aoOBCControllerOutput[x] /= covering;
212    OldLiBattCableConnectableToNewController = affords(
           feature_OldLiBattCableConnectableToNewController, ability_MoveNewController AND
           ability_MoveOldBattCableControllerOutput);
213        ability_MovePumpCableOutput =
214      pPumpOperator_aoPCControllerOutput.orientable[pitch_back] = TRUE AND
215      pPumpOperator_aoPCControllerOutput.orientable[pitch_forth] = TRUE AND
216      pPumpOperator_aoPCControllerOutput.orientable[yaw_left] = TRUE AND
217      pPumpOperator_aoPCControllerOutput.orientable[yaw_right] = TRUE AND
218      FORALL(x: translate): pPumpOperator_aoPCControllerOutput.translatable[x] = TRUE AND
219      FORALL(x: position): pPumpOperator_aoPCControllerOutput.positionable[x] = TRUE;
220        feature_PumpCableConnectableToNewController =
221          FORALL(x: directional): aoOCPumpInput.aoPCControllerOutput[x] /= covering AND
222          FORALL(x: directional): aoNCPumpInput.aoPCControllerOutput[x] /= covering AND
223          FORALL(x: directional): aoACPumpInput.aoPCControllerOutput[x] /= covering;
224    PumpCableConnectableToNewController = affords(feature_PumpCableConnectableToNewController,
           ability_MoveNewController AND ability_MovePumpCableOutput);
```

```
225      ability_MoveOldControllerBack =
226    pPumpOperator_mOldController.positionable[back] = TRUE;
227      ability_MovePumpCableControllerOutputBack =
228    pPumpOperator_aoPCControllerOutput.positionable[back] = TRUE;
229      feature_PumpCableDisconnectableFromOldController =
230        FORALL(x: directional): aoOCPumpInput.aoPCControllerOutput[x] = covering;
231  PumpCableDisconnectableFromOldController = affords(
         feature_PumpCableDisconnectableFromOldController, ability_MoveOldControllerBack AND
         ability_MovePumpCableControllerOutputBack);
232      ability_MoveAbCablePumpInputBack =
233    pPumpOperator_aoACPumpInput.positionable[back] = TRUE;
234      feature_PumpCableDisconnectableFromAbCable =
235        FORALL(x: directional): aoACPumpInput.aoPCControllerOutput[x] = covering;
236  PumpCableDisconnectableFromAbCable = affords(feature_PumpCableDisconnectableFromAbCable,
         ability_MoveAbCablePumpInputBack AND ability_MovePumpCableControllerOutputBack);
237
238    END;
239
240    HES: MODULE =
241      BEGIN
242
243        INPUT iLeadBattToOldController: userManual!tConnection
244        INPUT iLeadBattToNewController: userManual!tConnection
245        INPUT iLeadBattToYCable: userManual!tConnection
246        INPUT iYCableToOldController: userManual!tConnection
247        INPUT iOldLiBattCableToYCable: userManual!tConnection
248        INPUT iOldLiBattCableToOldController: userManual!tConnection
249        INPUT iOldLiBattCableToOldLiBatt: userManual!tConnection
250        INPUT iPumpCableToOldController: userManual!tConnection
251        INPUT iAbCableToOldController: userManual!tConnection
252        INPUT iNewLiBattCableToNewLiBatt: userManual!tConnection
253        INPUT iNewLiBattCableToNewController: userManual!tConnection
254        INPUT iPumpCableToOldAbCable: userManual!tConnection
255        INPUT iPumpCableToNewController: userManual!tConnection
256        INPUT iPermanentlyAttachedConnector: userManual!tPermAttachedConnectorStatus
257
258        OUTPUT aoConnectorPart1: aoConnectorPart1_rels
259        OUTPUT aoConnectorPart2: aoConnectorPart2_rels
260        OUTPUT aoNCBatteryInput: aoNCBatteryInput_rels
261        OUTPUT aoOCBatteryInput: aoOCBatteryInput_rels
262        OUTPUT aoOBBattCableInput: aoOBBattCableInput_rels
263        OUTPUT aoNBBattCableInput: aoNBBattCableInput_rels
264        OUTPUT aoYCableBatteryInput1: aoYCableBatteryInput1_rels
265        OUTPUT aoYCableBatteryInput2: aoYCableBatteryInput2_rels
266        OUTPUT aoOCPumpInput: aoOCPumpInput_rels
267        OUTPUT aoNCPumpInput: aoNCPumpInput_rels
268        OUTPUT aoACPumpInput: aoACPumpInput_rels
269
270        OUTPUT pPumpOperator_aoConnectorPart1: abilities
271        OUTPUT pPumpOperator_aoConnectorPart2: abilities
272        OUTPUT pPumpOperator_mNewController: abilities
273        OUTPUT pPumpOperator_aoNBCControllerOutput: abilities
274        OUTPUT pPumpOperator_aoLeadBattControllerOutput: abilities
275        OUTPUT pPumpOperator_mOldLiIonBattery: abilities
276        OUTPUT pPumpOperator_aoNBCBatteryOutput: abilities
277        OUTPUT pPumpOperator_mNewLiIonBattery: abilities
278        OUTPUT pPumpOperator_aoPCControllerOutput: abilities
279        OUTPUT pPumpOperator_mOldController: abilities
280        OUTPUT pPumpOperator_aoOBCControllerOutput: abilities
281
282        INITIALIZATION
283          (FORALL(y: position): pPumpOperator_aoConnectorPart1.positionable[y] = TRUE);
284          (FORALL(y: translate): pPumpOperator_aoConnectorPart1.translatable[y] = TRUE);
285          (FORALL(y: orient): pPumpOperator_aoConnectorPart1.orientable[y] = TRUE);
286
287          (FORALL(y: position): pPumpOperator_aoConnectorPart2.positionable[y] = TRUE);
288          (FORALL(y: translate): pPumpOperator_aoConnectorPart2 .translatable[y] = TRUE);
289          (FORALL(y: orient): pPumpOperator_aoConnectorPart2.orientable[y] = TRUE);
290
291          (FORALL(y: position): pPumpOperator_mNewController.positionable[y] = TRUE);
292          (FORALL(y: translate): pPumpOperator_mNewController.translatable[y] = TRUE);
293          (FORALL(y: orient): pPumpOperator_mNewController.orientable[y] = TRUE);
294
295          (FORALL(y: position): pPumpOperator_aoNBCControllerOutput.positionable[y] = TRUE);
296          (FORALL(y: translate): pPumpOperator_aoNBCControllerOutput.translatable[y] = TRUE);
297          (FORALL(y: orient): pPumpOperator_aoNBCControllerOutput.orientable[y] = TRUE);
298
299          (FORALL(y: position): pPumpOperator_aoLeadBattControllerOutput.positionable[y] = TRUE);
300          (FORALL(y: translate): pPumpOperator_aoLeadBattControllerOutput.translatable[y] = TRUE);
301          (FORALL(y: orient): pPumpOperator_aoLeadBattControllerOutput.orientable[y] = TRUE);
302
```

```
303         (FORALL(y: position): pPumpOperator_mOldLiIonBattery.positionable[y] = TRUE);
304         (FORALL(y: translate): pPumpOperator_mOldLiIonBattery.translatable[y] = TRUE);
305         (FORALL(y: orient): pPumpOperator_mOldLiIonBattery.orientable[y] = TRUE);
306
307         (FORALL(y: position): pPumpOperator_aoNBCBatteryOutput.positionable[y] = TRUE);
308         (FORALL(y: translate): pPumpOperator_aoNBCBatteryOutput.translatable[y] = TRUE);
309         (FORALL(y: orient): pPumpOperator_aoNBCBatteryOutput.orientable[y] = TRUE);
310
311         (FORALL(y: position): pPumpOperator_mNewLiIonBattery.positionable[y] = TRUE);
312         (FORALL(y: translate): pPumpOperator_mNewLiIonBattery.translatable[y] = TRUE);
313         (FORALL(y: orient): pPumpOperator_mNewLiIonBattery.orientable[y] = TRUE);
314
315         (FORALL(y: position): pPumpOperator_aoOBCControllerOutput.positionable[y] = TRUE);
316         (FORALL(y: translate): pPumpOperator_aoOBCControllerOutput.translatable[y] = TRUE);
317         (FORALL(y: orient): pPumpOperator_aoOBCControllerOutput.orientable[y] = TRUE);
318
319         (FORALL(y: position): pPumpOperator_aoPCControllerOutput.positionable[y] = TRUE);
320         (FORALL(y: translate): pPumpOperator_aoPCControllerOutput.translatable[y] = TRUE);
321         (FORALL(y: orient): pPumpOperator_aoPCControllerOutput.orientable[y] = TRUE);
322
323         (FORALL(y: position): pPumpOperator_mOldController.positionable[y] = TRUE);
324         (FORALL(y: translate): pPumpOperator_mOldController.translatable[y] = TRUE);
325         (FORALL(y: orient): pPumpOperator_mOldController.orientable[y] = TRUE);
326
327       LOCAL rels: relations
328       DEFINITION
329       aoOCPumpInput =
330         IF iPumpCableToOldController = userManual!Connected THEN
331           (#aoPCControllerOutput := [[x: directional]covering]#)
332         ELSE
333         (#aoPCControllerOutput := [[x: directional]disjoint_to]#)
334         ENDIF;
335
336       aoACPumpInput =
337         IF iPumpCableToOldAbCable = userManual!Connected THEN
338           (#aoPCControllerOutput := [[x: directional]covering]#)
339         ELSE
340           (#aoPCControllerOutput := [[x: directional]disjoint_to]#)
341         ENDIF;
342
343       aoNCPumpInput =
344         IF iPumpCableToNewController = userManual!Connected THEN
345           (#aoPCControllerOutput := [[x: directional]covering]#)
346         ELSE
347           (#aoPCControllerOutput := [[x: directional]disjoint_to]#)
348         ENDIF;
349
350       aoNCBatteryInput =
351         IF iLeadBattToNewController = userManual!Connected THEN
352           (#aoNBCControllerOutput := [[x: directional]disjoint_to],
353             aoLeadBattControllerOutput := [[x: directional]covering],
354             aoOBCControllerOutput := [[x: directional]disjoint_to],
355             aoYCControllerOutput := [[x: directional]disjoint_to]#)
356
357         ELSIF iNewLiBattCableToNewController = userManual!Connected THEN
358           (#aoNBCControllerOutput := [[x: directional]covering],
359             aoLeadBattControllerOutput := [[x: directional]disjoint_to],
360             aoOBCControllerOutput := [[x: directional]disjoint_to],
361             aoYCControllerOutput := [[x: directional]disjoint_to]#)
362
363         ELSE
364           (#aoNBCControllerOutput := [[x: directional]disjoint_to],
365             aoLeadBattControllerOutput := [[x: directional]disjoint_to],
366             aoOBCControllerOutput := [[x: directional]disjoint_to],
367             aoYCControllerOutput := [[x: directional]disjoint_to]#)
368         ENDIF;
369
370         aoOCBatteryInput =
371           IF iLeadBattToOldController = userManual!Connected THEN
372             (#aoNBCControllerOutput := [[x: directional]disjoint_to],
373               aoLeadBattControllerOutput := [[x: directional]covering],
374               aoOBCControllerOutput := [[x: directional]disjoint_to]#)
375
376           ELSIF iOldLiBattCableToOldController = userManual!Connected THEN
377             (#aoNBCControllerOutput := [[x: directional]disjoint_to],
378               aoLeadBattControllerOutput := [[x: directional]disjoint_to],
379               aoOBCControllerOutput := [[x: directional]covering]#)
380
381           ELSE
382             (#aoNBCControllerOutput := [[x: directional]disjoint_to],
383               aoLeadBattControllerOutput := [[x: directional]disjoint_to],
```

```
384            aoOBCControllerOutput := [[x: directional]disjoint_to]#)
385          ENDIF;
386
387        aoYCableBatteryInput1 IN
388          IF iOldLiBattCableToYCable = userManual!Connected THEN
389            {(#aoLeadBattControllerOutput := [[x: directional]disjoint_to],
390              aoNBCControllerOutput       := [[x: directional]disjoint_to],
391              aoOBCControllerOutput       := [[x: directional]covering]#),
392
393            (#aoLeadBattControllerOutput  := [[x: directional]disjoint_to],
394              aoNBCControllerOutput       := [[x: directional]disjoint_to],
395              aoOBCControllerOutput       := [[x: directional]disjoint_to]#)}
396
397          ELSIF iLeadBattToYCable = userManual!Connected THEN
398            {(#aoLeadBattControllerOutput := [[x: directional]covering],
399              aoNBCControllerOutput       := [[x: directional]disjoint_to],
400              aoOBCControllerOutput       := [[x: directional]disjoint_to]#),
401
402            (#aoLeadBattControllerOutput  := [[x: directional]disjoint_to],
403              aoNBCControllerOutput       := [[x: directional]disjoint_to],
404              aoOBCControllerOutput       := [[x: directional]disjoint_to]#)}
405
406        ELSE
407        {(#aoLeadBattControllerOutput := [[x: directional]disjoint_to],
408          aoNBCControllerOutput := [[x: directional]disjoint_to],
409          aoOBCControllerOutput := [[x: directional]disjoint_to]#)}
410        ENDIF;
411
412        aoYCableBatteryInput2 =
413          IF iOldLiBattCableToYCable = userManual!Connected AND
414            FORALL(x: directional): aoYCableBatteryInput1.aoOBCControllerOutput[x] = disjoint_to
415          THEN
416          (#aoLeadBattControllerOutput := [[x: directional]disjoint_to],
417            aoNBCControllerOutput := [[x: directional]disjoint_to],
418            aoOBCControllerOutput := [[x: directional]covering]#)
419
420          ELSIF iOldLiBattCableToYCable = userManual!Connected AND
421            FORALL(x: directional): aoYCableBatteryInput1.aoOBCControllerOutput[x] = covering
422          THEN
423          (#aoLeadBattControllerOutput := [[x: directional]disjoint_to],
424            aoNBCControllerOutput := [[x: directional]disjoint_to],
425            aoOBCControllerOutput := [[x: directional]disjoint_to]#)
426
427          ELSIF iLeadBattToYCable = userManual!Connected AND
428            FORALL(x: directional): aoYCableBatteryInput1.aoLeadBattControllerOutput[x] =
                 disjoint_to
429          THEN
430          (#aoLeadBattControllerOutput := [[x: directional]covering],
431            aoNBCControllerOutput := [[x: directional]disjoint_to],
432            aoOBCControllerOutput := [[x: directional]disjoint_to]#)
433
434          ELSIF iLeadBattToYCable = userManual!Connected AND
435            FORALL(x: directional): aoYCableBatteryInput1.aoLeadBattControllerOutput[x] =
                 covering
436          THEN
437          (#aoLeadBattControllerOutput := [[x: directional]disjoint_to],
438            aoNBCControllerOutput := [[x: directional]disjoint_to],
439            aoOBCControllerOutput := [[x: directional]disjoint_to]#)
440
441        ELSE
442        (#aoLeadBattControllerOutput := [[x: directional]disjoint_to],
443          aoNBCControllerOutput := [[x: directional]disjoint_to],
444          aoOBCControllerOutput := [[x: directional]disjoint_to]#)
445        ENDIF;
446
447        aoNBBattCableInput =
448          IF iNewLiBattCableToNewLiBatt = userManual!Connected THEN
449            (#aoNBCBatteryOutput   := [[x: directional]covering],
450             aoOBCBatteryOutput    := [[x: directional]disjoint_to]#)
451          ELSE
452            (#aoNBCBatteryOutput   := [[x: directional]disjoint_to],
453             aoOBCBatteryOutput    := [[x: directional]disjoint_to]#)
454          ENDIF;
455
456        aoOBBattCableInput =
457          IF iOldLiBattCableToOldLiBatt = userManual!Connected THEN
458            (#aoNBCBatteryOutput   := [[x: directional]disjoint_to],
459             aoOBCBatteryOutput    := [[x: directional]covering]#)
460          ELSE
461            (#aoNBCBatteryOutput   := [[x: directional]disjoint_to],
462             aoOBCBatteryOutput    := [[x: directional]disjoint_to]#)
```

```
463             ENDIF;
464
465           aoConnectorPart1 =
466             IF iPermanentlyAttachedConnector = userManual!Assembled THEN
467               (#aoConnectorPart2 := rels
468                                  WITH [left_of] := disjoint_to WITH [right_of] := disjoint_to
469                                  WITH [top_of] := disjoint_to WITH [bottom_of] := disjoint_to
470                                  WITH [front_of] := covering WITH [back_of] := disjoint_to#)
471             ELSE
472               (#aoConnectorPart2 := [[x: directional]disjoint_to]#)
473             ENDIF;
474
475             aoConnectorPart2 =
476               IF iPermanentlyAttachedConnector = userManual!Assembled THEN
477                 (#aoConnectorPart1 := rels
478                                    WITH [left_of] := disjoint_to WITH [right_of] := disjoint_to
479                                    WITH [top_of] := disjoint_to WITH [bottom_of] := disjoint_to
480                                    WITH [front_of] := disjoint_to WITH [back_of] := covering#)
481               ELSE
482                 (#aoConnectorPart1 := [[x: directional]disjoint_to]#)
483               ENDIF;
484     END;
485     hes: MODULE = affordance || HES;
486 END
```

H.2.4 Signifier Model

```
1  bigsis: CONTEXT =
2  BEGIN
3    PumpSpeed: TYPE = {PumpSpeedNotSignified, Stopped, Low, Lowest, Medium, High, Highest};
4    PowerSupplied: TYPE = {PowerSuppliedNotSignified, ZeroUnits, OneUnit, TwoUnits, ThreeUnits,
         FourUnits,
5          FiveUnits, SixUnits, SevenUnits, EightUnits, NineUnits, TenUnits, TooHigh};
6    COLORS: TYPE = {green,amber,red,noColor,white};
7    LABELS: TYPE = {one,two,three,four,five,six,seven,eight,nine,ten,HIGH,noLabel};
8    PATTERNS: TYPE = {POWER_TOO_HIGH,PUMP_STOPPED,noPattern};
9    LEVELS: TYPE = {loud,noLevel};
10   Colors_signify: TYPE = [#Colored: COLORS, PowerSupplied: PowerSupplied, PumpSpeed: PumpSpeed#];
11   Labels_signify: TYPE = [#Labeled: LABELS, PowerSupplied: PowerSupplied, PumpSpeed: PumpSpeed#];
12   Volumes_signify: TYPE = [#Level: LEVELS, PowerSupplied: PowerSupplied, PumpSpeed: PumpSpeed#];
13   aPatterns_signify: TYPE = [#Pattern: PATTERNS, PowerSupplied: PowerSupplied, PumpSpeed:
         PumpSpeed#];
14   PowerIndicators_signifiers: TYPE = [#Color: Colors_signify, Label: Labels_signify, Volume:
         Volumes_signify, aPattern: aPatterns_signify#];
15   PumpStoppedAlarm_signifiers: TYPE = [#Color: Colors_signify, Volume: Volumes_signify, aPattern:
          aPatterns_signify#];
16   SpeedSettingKnob_signifiers: TYPE = [#Color: Colors_signify, Label: Labels_signify#];
17   Doc_PowerIndicators_signifiers: TYPE = [#Color: Colors_signify, Label: Labels_signify, Volume:
         Volumes_signify, aPattern: aPatterns_signify#];
18   Doc_PumpStoppedAlarm_signifiers: TYPE = [#Color: Colors_signify, Volume: Volumes_signify,
         aPattern: aPatterns_signify#];
19   Doc_SpeedSettingKnob_signifiers: TYPE = [#Color: Colors_signify, Label: Labels_signify#];
20 signifiers: MODULE =
21  BEGIN
22    LOCAL PowerIndicators: PowerIndicators_signifiers
23    LOCAL PumpStoppedAlarm: PumpStoppedAlarm_signifiers
24    LOCAL SpeedSettingKnob: SpeedSettingKnob_signifiers
25    LOCAL Doc_PowerIndicators: Doc_PowerIndicators_signifiers
26    LOCAL Doc_PumpStoppedAlarm: Doc_PumpStoppedAlarm_signifiers
27    LOCAL Doc_SpeedSettingKnob: Doc_SpeedSettingKnob_signifiers
28    INPUT iPage: {x: INTEGER | x >= 0 AND x <= 29}
29    INPUT iNewControllerAlarmBatteryCap: userManual!tAlarmBatteryCap
30    INPUT iOldControllerAlarmBatteryCap: userManual!tAlarmBatteryCap
31    INPUT iAlarm: discreteDevice!Alarms
32    INPUT iPowerLight: discreteDevice!PowerLights
33    INPUT iSpeedSetting: discreteDevice!SpeedSettings
34
35    INITIALIZATION [
36      iAlarm = discreteDevice!NoAlarm -->
37        PowerIndicators.Color.Colored = green;
38        PumpStoppedAlarm.Color.Colored = noColor;
39        PumpStoppedAlarm.Volume.Level = noLevel;
40        PowerIndicators.Volume.Level = noLevel;
41        PumpStoppedAlarm.aPattern.Pattern = noPattern;
42        PowerIndicators.aPattern.Pattern = noPattern;
43        SpeedSettingKnob.Color.Colored = white;
44        SpeedSettingKnob.Label.Labeled =
45         IF iSpeedSetting = 1 THEN one
46          ELSIF iSpeedSetting = 2 THEN two
47          ELSIF iSpeedSetting = 3 THEN three
```

```
48          ELSIF iSpeedSetting = 4 THEN four
49          ELSIF iSpeedSetting = 5 THEN five
50         ELSE noLabel ENDIF;
51        PowerIndicators.Label.Labeled =
52         IF iPowerLight = 0 THEN noLabel
53            ELSIF iPowerLight = 1 THEN one
54            ELSIF iPowerLight = 2 THEN two
55            ELSIF iPowerLight = 3 THEN three
56            ELSIF iPowerLight = 4 THEN four
57            ELSIF iPowerLight = 5 THEN five
58            ELSIF iPowerLight = 6 THEN six
59            ELSIF iPowerLight = 7 THEN seven
60            ELSIF iPowerLight = 8 THEN eight
61            ELSIF iPowerLight = 9 THEN nine
62            ELSIF iPowerLight = 10 THEN ten
63           ELSE HIGH ENDIF;
64       []iAlarm = discreteDevice!PumpStopped -->
65          PowerIndicators.Color.Colored = IF iPowerLight = 0 THEN noColor ELSE green ENDIF;
66          PumpStoppedAlarm.Color.Colored = red;
67          PumpStoppedAlarm.Volume.Level = IF (iNewControllerAlarmBatteryCap = userManual!Tightened
                 OR iOldControllerAlarmBatteryCap = userManual!Tightened) THEN loud ELSE noLevel
                ENDIF;
68          PowerIndicators.Volume.Level = noLevel;
69          PumpStoppedAlarm.aPattern.Pattern = IF (iNewControllerAlarmBatteryCap = userManual!
                Tightened XOR iOldControllerAlarmBatteryCap = userManual!Tightened) THEN
                PUMP_STOPPED ELSE noPattern ENDIF;
70          PowerIndicators.aPattern.Pattern = noPattern;
71          SpeedSettingKnob.Color.Colored = noColor;
72          SpeedSettingKnob.Label.Labeled = noLabel;
73          PowerIndicators.Label.Labeled =
74           IF iPowerLight = 0 THEN noLabel
75              ELSIF iPowerLight = 1 THEN one
76              ELSIF iPowerLight = 2 THEN two
77              ELSIF iPowerLight = 3 THEN three
78              ELSIF iPowerLight = 4 THEN four
79              ELSIF iPowerLight = 5 THEN five
80              ELSIF iPowerLight = 6 THEN six
81              ELSIF iPowerLight = 7 THEN seven
82              ELSIF iPowerLight = 8 THEN eight
83              ELSIF iPowerLight = 9 THEN nine
84              ELSIF iPowerLight = 10 THEN ten
85             ELSE HIGH ENDIF;
86        []iAlarm = discreteDevice!HighPower -->
87            PowerIndicators.Color.Colored = amber;
88            PumpStoppedAlarm.Color.Colored = noColor;
89            PowerIndicators.Volume.Level = IF (iNewControllerAlarmBatteryCap = userManual!
                  Tightened OR iOldControllerAlarmBatteryCap = userManual!Tightened) THEN loud ELSE
                   noLevel ENDIF;
90            PumpStoppedAlarm.Volume.Level = noLevel;
91            PumpStoppedAlarm.aPattern.Pattern = noPattern;
92            PowerIndicators.aPattern.Pattern = IF (iNewControllerAlarmBatteryCap = userManual!
                  Tightened XOR iOldControllerAlarmBatteryCap = userManual!Tightened) THEN
                  POWER_TOO_HIGH ELSE noPattern ENDIF;
93            SpeedSettingKnob.Color.Colored = white;
94            SpeedSettingKnob.Label.Labeled =
95              IF iSpeedSetting = 1 THEN one
96               ELSIF iSpeedSetting = 2 THEN two
97               ELSIF iSpeedSetting = 3 THEN three
98               ELSIF iSpeedSetting = 4 THEN four
99               ELSIF iSpeedSetting = 5 THEN five
100             ELSE noLabel ENDIF;
101           PowerIndicators.Label.Labeled = HIGH;
102        ];
103     PowerIndicators.Color.PowerSupplied = IF PowerIndicators.Color.Colored = green THEN
            PowerIndicators.Label.PowerSupplied ELSIF PowerIndicators.Color.Colored = amber THEN
            TooHigh ELSE PowerSuppliedNotSignified ENDIF;
104     PowerIndicators.Label.PowerSupplied = IF PowerIndicators.Label.Labeled = one THEN OneUnit
            ELSIF PowerIndicators.Label.Labeled = two THEN TwoUnits ELSIF PowerIndicators.Label.
            Labeled = three THEN ThreeUnits ELSIF PowerIndicators.Label.Labeled = four THEN
            FourUnits ELSIF PowerIndicators.Label.Labeled = five THEN FiveUnits ELSIF
            PowerIndicators.Label.Labeled = six THEN SixUnits ELSIF PowerIndicators.Label.Labeled =
            seven THEN SevenUnits ELSIF PowerIndicators.Label.Labeled = eight THEN EightUnits ELSIF
            PowerIndicators.Label.Labeled = nine THEN NineUnits ELSIF PowerIndicators.Label.Labeled
            = ten THEN TenUnits ELSIF PowerIndicators.Label.Labeled = HIGH THEN TooHigh ELSE
            PowerSuppliedNotSignified ENDIF;
105     PowerIndicators.Volume.PowerSupplied = IF PowerIndicators.Volume.Level = loud THEN
            PowerIndicators.aPattern.PowerSupplied ELSE PowerSuppliedNotSignified ENDIF;
106     PowerIndicators.aPattern.PowerSupplied = IF PowerIndicators.aPattern.Pattern = POWER_TOO_HIGH
             THEN TooHigh ELSE PowerSuppliedNotSignified ENDIF;
107     PumpStoppedAlarm.Color.PumpSpeed = IF PumpStoppedAlarm.Color.Colored = red THEN Stopped ELSIF
             PumpStoppedAlarm.Color.Colored = noColor THEN SpeedSettingKnob.Label.PumpSpeed ELSE
```

```
              PumpSpeedNotSignified ENDIF;
108       PumpStoppedAlarm.Volume.PumpSpeed = IF PumpStoppedAlarm.Volume.Level = loud THEN
              PumpStoppedAlarm.aPattern.PumpSpeed ELSE PumpSpeedNotSignified ENDIF;
109       PumpStoppedAlarm.aPattern.PumpSpeed = IF PumpStoppedAlarm.aPattern.Pattern = PUMP_STOPPED
              THEN Stopped ELSE PumpSpeedNotSignified ENDIF;
110       SpeedSettingKnob.Color.PumpSpeed = IF SpeedSettingKnob.Color.Colored = white THEN
              SpeedSettingKnob.Label.PumpSpeed ELSIF SpeedSettingKnob.Color.Colored = noColor THEN
              PumpStoppedAlarm.Color.PumpSpeed ELSE PumpSpeedNotSignified ENDIF;
111       SpeedSettingKnob.Label.PumpSpeed = IF SpeedSettingKnob.Label.Labeled = noLabel THEN
              PumpStoppedAlarm.Color.PumpSpeed ELSIF SpeedSettingKnob.Label.Labeled = one THEN Lowest
              ELSIF SpeedSettingKnob.Label.Labeled = two THEN Low ELSIF SpeedSettingKnob.Label.Labeled
               = three THEN Medium ELSIF SpeedSettingKnob.Label.Labeled = four THEN High ELSIF
              SpeedSettingKnob.Label.Labeled = five THEN Highest ELSE PumpSpeedNotSignified ENDIF;
112       Doc_PowerIndicators.Color.PowerSupplied = IF PowerIndicators.Color.Colored = green AND iPage
              = 8 THEN Doc_PowerIndicators.Label.PowerSupplied ELSIF PowerIndicators.Color.Colored =
              noColor AND iPage = 10 THEN ZeroUnits ELSIF PowerIndicators.Color.Colored = amber AND
              iPage = 10 THEN TooHigh ELSE PowerSuppliedNotSignified ENDIF;
113       Doc_PowerIndicators.Color.PumpSpeed = IF PowerIndicators.Color.Colored = noColor THEN Stopped
               ELSE PumpSpeedNotSignified ENDIF;
114       Doc_PowerIndicators.Label.PowerSupplied = IF PowerIndicators.Label.Labeled = noLabel AND
              iPage = 10 THEN Doc_PowerIndicators.Color.PowerSupplied ELSIF PowerIndicators.Label.
              Labeled = HIGH AND iPage = 10 THEN TooHigh ELSIF PowerIndicators.Label.Labeled = one AND
               iPage = 8 THEN OneUnit ELSIF PowerIndicators.Label.Labeled = two AND iPage = 8 THEN
              TwoUnits ELSIF PowerIndicators.Label.Labeled = three AND iPage = 8 THEN ThreeUnits ELSIF
               PowerIndicators.Label.Labeled = four AND iPage = 8 THEN FourUnits ELSIF PowerIndicators
              .Label.Labeled = five AND iPage = 8 THEN FiveUnits ELSIF PowerIndicators.Label.Labeled =
               six AND iPage = 8 THEN SixUnits ELSIF PowerIndicators.Label.Labeled = seven AND iPage =
               8 THEN SevenUnits ELSIF PowerIndicators.Label.Labeled = eight AND iPage = 8 THEN
              EightUnits ELSIF PowerIndicators.Label.Labeled = nine AND iPage = 8 THEN NineUnits ELSIF
               PowerIndicators.Label.Labeled = ten AND iPage = 8 THEN TenUnits ELSE
              PowerSuppliedNotSignified ENDIF;
115       Doc_PowerIndicators.Volume.PowerSupplied = IF PowerIndicators.Volume.Level = loud THEN
              Doc_PowerIndicators.aPattern.PowerSupplied ELSE PowerSuppliedNotSignified ENDIF;
116       Doc_PowerIndicators.aPattern.PowerSupplied = IF PowerIndicators.aPattern.Pattern =
              POWER_TOO_HIGH AND iPage = 10 THEN TooHigh ELSE PowerSuppliedNotSignified ENDIF;
117       Doc_PumpStoppedAlarm.Color.PumpSpeed = IF PumpStoppedAlarm.Color.Colored = red AND iPage = 10
               THEN Stopped ELSIF PumpStoppedAlarm.Color.Colored = noColor THEN Doc_SpeedSettingKnob.
              Label.PumpSpeed ELSE PumpSpeedNotSignified ENDIF;
118       Doc_PumpStoppedAlarm.Volume.PumpSpeed = IF PumpStoppedAlarm.Volume.Level = loud AND iPage =
              10 THEN Doc_PumpStoppedAlarm.aPattern.PumpSpeed ELSE PumpSpeedNotSignified ENDIF;
119       Doc_PumpStoppedAlarm.aPattern.PumpSpeed = IF PumpStoppedAlarm.aPattern.Pattern = PUMP_STOPPED
               AND iPage = 10 THEN Stopped ELSE PumpSpeedNotSignified ENDIF;
120       Doc_SpeedSettingKnob.Color.PumpSpeed = IF SpeedSettingKnob.Color.Colored = white THEN
              Doc_SpeedSettingKnob.Label.PumpSpeed ELSIF SpeedSettingKnob.Color.Colored = noColor THEN
               Doc_PumpStoppedAlarm.Color.PumpSpeed ELSE PumpSpeedNotSignified ENDIF;
121       Doc_SpeedSettingKnob.Label.PumpSpeed = IF SpeedSettingKnob.Label.Labeled = noLabel AND iPage
              = 10 THEN Stopped ELSIF SpeedSettingKnob.Label.Labeled = one AND iPage = 8 THEN Lowest
              ELSIF SpeedSettingKnob.Label.Labeled = two AND iPage = 8 THEN Low ELSIF SpeedSettingKnob
              .Label.Labeled = three AND iPage = 8 THEN Medium ELSIF SpeedSettingKnob.Label.Labeled =
              four AND iPage = 8 THEN High ELSIF SpeedSettingKnob.Label.Labeled = five AND iPage = 8
              THEN Highest ELSE PumpSpeedNotSignified ENDIF;
122
123     TRANSITION
124     [
125     iAlarm' = discreteDevice!NoAlarm -->
126        PowerIndicators'.Color.Colored = green;
127        PumpStoppedAlarm'.Color.Colored = noColor;
128        PumpStoppedAlarm'.Volume.Level = noLevel;
129       PowerIndicators'.Volume.Level = noLevel;
130        PumpStoppedAlarm'.aPattern.Pattern = noPattern;
131        PowerIndicators'.aPattern.Pattern = noPattern;
132        SpeedSettingKnob'.Color.Colored = white;
133        SpeedSettingKnob'.Label.Labeled =
134         IF iSpeedSetting' = 1 THEN one
135          ELSIF iSpeedSetting' = 2 THEN two
136          ELSIF iSpeedSetting' = 3 THEN three
137          ELSIF iSpeedSetting' = 4 THEN four
138         ELSIF iSpeedSetting' = 5 THEN five
139         ELSE noLabel ENDIF;
140        PowerIndicators'.Label.Labeled =
141         IF iPowerLight' = 0 THEN noLabel
142            ELSIF iPowerLight' = 1 THEN one
143            ELSIF iPowerLight' = 2 THEN two
144            ELSIF iPowerLight' = 3 THEN three
145            ELSIF iPowerLight' = 4 THEN four
146            ELSIF iPowerLight' = 5 THEN five
147            ELSIF iPowerLight' = 6 THEN six
148            ELSIF iPowerLight' = 7 THEN seven
149            ELSIF iPowerLight' = 8 THEN eight
150            ELSIF iPowerLight' = 9 THEN nine
151            ELSIF iPowerLight' = 10 THEN ten
```

```
152        ELSE HIGH ENDIF;
153    []iAlarm' = discreteDevice!PumpStopped -->
154        PowerIndicators'.Color.Colored = IF iPowerLight' = 0 THEN noColor ELSE green ENDIF;
155        PumpStoppedAlarm'.Color.Colored = red;
156        PumpStoppedAlarm'.Volume.Level = IF (iNewControllerAlarmBatteryCap' = userManual!Tightened
               OR iOldControllerAlarmBatteryCap' = userManual!Tightened) THEN loud ELSE noLevel
               ENDIF;
157        PowerIndicators'.Volume.Level = noLevel;
158        PumpStoppedAlarm'.aPattern.Pattern = IF (iNewControllerAlarmBatteryCap' = userManual!
               Tightened XOR iOldControllerAlarmBatteryCap' = userManual!Tightened) THEN
               PUMP_STOPPED ELSE noPattern ENDIF;
159        PowerIndicators'.aPattern.Pattern = noPattern;
160        SpeedSettingKnob'.Color.Colored = noColor;
161        SpeedSettingKnob'.Label.Labeled = noLabel;
162        PowerIndicators'.Label.Labeled =
163         IF iPowerLight' = 0 THEN noLabel
164            ELSIF iPowerLight' = 1 THEN one
165            ELSIF iPowerLight' = 2 THEN two
166            ELSIF iPowerLight' = 3 THEN three
167            ELSIF iPowerLight' = 4 THEN four
168            ELSIF iPowerLight' = 5 THEN five
169            ELSIF iPowerLight' = 6 THEN six
170            ELSIF iPowerLight' = 7 THEN seven
171            ELSIF iPowerLight' = 8 THEN eight
172            ELSIF iPowerLight' = 9 THEN nine
173            ELSIF iPowerLight' = 10 THEN ten
174           ELSE HIGH ENDIF;
175      []iAlarm' = discreteDevice!HighPower -->
176          PowerIndicators'.Color.Colored = amber;
177          PumpStoppedAlarm'.Color.Colored = noColor;
178          PowerIndicators'.Volume.Level = IF (iNewControllerAlarmBatteryCap = userManual!Tightened
                 OR iOldControllerAlarmBatteryCap = userManual!Tightened) THEN loud ELSE noLevel
                 ENDIF;
179          PumpStoppedAlarm'.Volume.Level = noLevel;
180          PumpStoppedAlarm'.aPattern.Pattern = noPattern;
181          PowerIndicators'.aPattern.Pattern = IF (iNewControllerAlarmBatteryCap = userManual!
                 Tightened XOR iOldControllerAlarmBatteryCap = userManual!Tightened) THEN
                 POWER_TOO_HIGH ELSE noPattern ENDIF;
182          SpeedSettingKnob'.Color.Colored = white;
183         SpeedSettingKnob'.Label.Labeled =
184           IF iSpeedSetting' = 1 THEN one
185            ELSIF iSpeedSetting' = 2 THEN two
186            ELSIF iSpeedSetting' = 3 THEN three
187            ELSIF iSpeedSetting' = 4 THEN four
188            ELSIF iSpeedSetting' = 5 THEN five
189           ELSE noLabel ENDIF;
190          PowerIndicators'.Label.Labeled = HIGH;
191       ];
192     PowerIndicators'.Color.PowerSupplied = IF PowerIndicators'.Color.Colored = green THEN
            PowerIndicators'.Label.PowerSupplied ELSIF PowerIndicators'.Color.Colored = amber THEN
            TooHigh ELSE PowerSuppliedNotSignified ENDIF;
193     PowerIndicators'.Label.PowerSupplied = IF PowerIndicators'.Label.Labeled = one THEN OneUnit
            ELSIF PowerIndicators'.Label.Labeled = two THEN TwoUnits ELSIF PowerIndicators'.Label.
            Labeled = three THEN ThreeUnits ELSIF PowerIndicators'.Label.Labeled = four THEN
            FourUnits ELSIF PowerIndicators'.Label.Labeled = five THEN FiveUnits ELSIF
            PowerIndicators'.Label.Labeled = six THEN SixUnits ELSIF PowerIndicators'.Label.Labeled
            = seven THEN SevenUnits ELSIF PowerIndicators'.Label.Labeled = eight THEN EightUnits
            ELSIF PowerIndicators'.Label.Labeled = nine THEN NineUnits ELSIF PowerIndicators'.Label.
            Labeled = ten THEN TenUnits ELSIF PowerIndicators'.Label.Labeled = HIGH THEN TooHigh
            ELSE PowerSuppliedNotSignified ENDIF;
194     PowerIndicators'.Volume.PowerSupplied = IF PowerIndicators'.Volume.Level = loud THEN
            PowerIndicators'.aPattern.PowerSupplied ELSE PowerSuppliedNotSignified ENDIF;
195     PowerIndicators'.aPattern.PowerSupplied = IF PowerIndicators'.aPattern.Pattern =
            POWER_TOO_HIGH THEN TooHigh ELSE PowerSuppliedNotSignified ENDIF;
196     PumpStoppedAlarm'.Color.PumpSpeed = IF PumpStoppedAlarm'.Color.Colored = red THEN Stopped
            ELSIF PumpStoppedAlarm'.Color.Colored = noColor THEN SpeedSettingKnob'.Label.PumpSpeed
            ELSE PumpSpeedNotSignified ENDIF;
197     PumpStoppedAlarm'.Volume.PumpSpeed = IF PumpStoppedAlarm'.Volume.Level = loud THEN
            PumpStoppedAlarm'.aPattern.PumpSpeed ELSE PumpSpeedNotSignified ENDIF;
198     PumpStoppedAlarm'.aPattern.PumpSpeed = IF PumpStoppedAlarm'.aPattern.Pattern = PUMP_STOPPED
            THEN Stopped ELSE PumpSpeedNotSignified ENDIF;
199     SpeedSettingKnob'.Color.PumpSpeed = IF SpeedSettingKnob'.Color.Colored = white THEN
            SpeedSettingKnob'.Label.PumpSpeed ELSIF SpeedSettingKnob'.Color.Colored = noColor THEN
            PumpStoppedAlarm'.Color.PumpSpeed ELSE PumpSpeedNotSignified ENDIF;
200     SpeedSettingKnob'.Label.PumpSpeed = IF SpeedSettingKnob'.Label.Labeled = noLabel THEN
            PumpStoppedAlarm'.Color.PumpSpeed ELSIF SpeedSettingKnob'.Label.Labeled = one THEN
            Lowest ELSIF SpeedSettingKnob'.Label.Labeled = two THEN Low ELSIF SpeedSettingKnob'.
            Label.Labeled = three THEN Medium ELSIF SpeedSettingKnob'.Label.Labeled = four THEN High
             ELSIF SpeedSettingKnob'.Label.Labeled = five THEN Highest ELSE PumpSpeedNotSignified
            ENDIF;
201     Doc_PowerIndicators'.Color.PowerSupplied = IF PowerIndicators'.Color.Colored = green AND
```

```
          iPage = 8 THEN Doc_PowerIndicators'.Label.PowerSupplied ELSIF PowerIndicators'.Color.
          Colored = noColor AND iPage = 10 THEN ZeroUnits ELSIF PowerIndicators'.Color.Colored =
          amber AND iPage = 10 THEN TooHigh ELSE PowerSuppliedNotSignified ENDIF;
202      Doc_PowerIndicators'.Color.PumpSpeed = IF PowerIndicators'.Color.Colored = noColor THEN
          Stopped ELSE PumpSpeedNotSignified ENDIF;
203      Doc_PowerIndicators'.Label.PowerSupplied = IF PowerIndicators'.Label.Labeled = noLabel AND
          iPage = 10 THEN Doc_PowerIndicators'.Color.PowerSupplied ELSIF PowerIndicators'.Label.
          Labeled = HIGH AND iPage = 10 THEN TooHigh ELSIF PowerIndicators'.Label.Labeled = one
          AND iPage = 8 THEN OneUnit ELSIF PowerIndicators'.Label.Labeled = two AND iPage = 8 THEN
           TwoUnits ELSIF PowerIndicators'.Label.Labeled = three AND iPage = 8 THEN ThreeUnits
          ELSIF PowerIndicators'.Label.Labeled = four AND iPage = 8 THEN FourUnits ELSIF
          PowerIndicators'.Label.Labeled = five AND iPage = 8 THEN FiveUnits ELSIF PowerIndicators
          '.Label.Labeled = six AND iPage = 8 THEN SixUnits ELSIF PowerIndicators'.Label.Labeled =
           seven AND iPage = 8 THEN SevenUnits ELSIF PowerIndicators'.Label.Labeled = eight AND
          iPage = 8 THEN EightUnits ELSIF PowerIndicators'.Label.Labeled = nine AND iPage = 8 THEN
           NineUnits ELSIF PowerIndicators'.Label.Labeled = ten AND iPage = 8 THEN TenUnits ELSE
          PowerSuppliedNotSignified ENDIF;
204      Doc_PowerIndicators'.Volume.PowerSupplied = IF PowerIndicators'.Volume.Level = loud THEN
          Doc_PowerIndicators'.aPattern.PowerSupplied ELSE PowerSuppliedNotSignified ENDIF;
205      Doc_PowerIndicators'.aPattern.PowerSupplied = IF PowerIndicators'.aPattern.Pattern =
          POWER_TOO_HIGH AND iPage = 10 THEN TooHigh ELSE PowerSuppliedNotSignified ENDIF;
206      Doc_PumpStoppedAlarm'.Color.PumpSpeed = IF PumpStoppedAlarm'.Color.Colored = red AND iPage =
          10 THEN Stopped ELSIF PumpStoppedAlarm'.Color.Colored = noColor AND iPage = 10 THEN
          Doc_SpeedSettingKnob'.Label.PumpSpeed ELSE PumpSpeedNotSignified ENDIF;
207      Doc_PumpStoppedAlarm'.Volume.PumpSpeed = IF PumpStoppedAlarm'.Volume.Level = loud AND iPage =
           10 THEN Doc_PumpStoppedAlarm'.aPattern.PumpSpeed ELSE PumpSpeedNotSignified ENDIF;
208      Doc_PumpStoppedAlarm'.aPattern.PumpSpeed = IF PumpStoppedAlarm'.aPattern.Pattern =
          PUMP_STOPPED AND iPage = 10 THEN Stopped ELSE PumpSpeedNotSignified ENDIF;
209      Doc_SpeedSettingKnob'.Color.PumpSpeed = IF SpeedSettingKnob'.Color.Colored = white THEN
          Doc_SpeedSettingKnob'.Label.PumpSpeed ELSIF SpeedSettingKnob'.Color.Colored = noColor
          THEN Doc_PumpStoppedAlarm'.Color.PumpSpeed ELSE PumpSpeedNotSignified ENDIF;
210      Doc_SpeedSettingKnob'.Label.PumpSpeed = IF SpeedSettingKnob'.Label.Labeled = noLabel AND
          iPage = 10 THEN Stopped ELSIF SpeedSettingKnob'.Label.Labeled = one AND iPage = 8 THEN
          Lowest ELSIF SpeedSettingKnob'.Label.Labeled = two AND iPage = 8 THEN Low ELSIF
          SpeedSettingKnob'.Label.Labeled = three AND iPage = 8 THEN Medium ELSIF SpeedSettingKnob
          '.Label.Labeled = four AND iPage = 8 THEN High ELSIF SpeedSettingKnob'.Label.Labeled =
          five AND iPage = 8 THEN Highest ELSE PumpSpeedNotSignified ENDIF;
211
212     OUTPUT Visually_Signified_PumpSpeed: PumpSpeed
213     OUTPUT Audibly_Signified_PumpSpeed: PumpSpeed
214     OUTPUT Documented_PumpSpeed: PumpSpeed
215     OUTPUT Visually_Signified_PowerSupplied: PowerSupplied
216     OUTPUT Audibly_Signified_PowerSupplied: PowerSupplied
217     OUTPUT Documented_PowerSupplied: PowerSupplied
218
219   DEFINITION
220       Visually_Signified_PumpSpeed IN {PumpStoppedAlarm.Color.PumpSpeed, SpeedSettingKnob.Color.
           PumpSpeed, SpeedSettingKnob.Label.PumpSpeed};
221       Visually_Signified_PowerSupplied IN {PowerIndicators.Color.PowerSupplied, PowerIndicators.
           Label.PowerSupplied};
222       Audibly_Signified_PumpSpeed IN {PumpStoppedAlarm.Volume.PumpSpeed, PumpStoppedAlarm.aPattern
           .PumpSpeed};
223       Audibly_Signified_PowerSupplied IN {PowerIndicators.Volume.PowerSupplied, PowerIndicators.
           aPattern.PowerSupplied};
224       Documented_PumpSpeed IN {Doc_PowerIndicators.Color.PumpSpeed, Doc_PumpStoppedAlarm.Color.
           PumpSpeed, Doc_PumpStoppedAlarm.Volume.PumpSpeed, Doc_PumpStoppedAlarm.aPattern.
           PumpSpeed, Doc_SpeedSettingKnob.Color.PumpSpeed, Doc_SpeedSettingKnob.Label.PumpSpeed};
225       Documented_PowerSupplied IN {Doc_PowerIndicators.Color.PowerSupplied, Doc_PowerIndicators.
           Label.PowerSupplied, Doc_PowerIndicators.Volume.PowerSupplied, Doc_PowerIndicators.
           aPattern.PowerSupplied};
226   END;
227 END
```

H.2.5 Discrete Device

```
1  discreteDevice: CONTEXT =
2    BEGIN
3    BatteryLights: TYPE = [0..5];
4    PowerLights: TYPE = [0..11];
5    SpeedSettings: TYPE = [1..5];
6    Alarms: TYPE = {PumpStopped, HighPower, NoAlarm};
7
8    system: MODULE = % EOFM handshake module
9    BEGIN
10     OUTPUT ready: BOOLEAN
11     INPUT submitted: BOOLEAN
12
13     INITIALIZATION
14       ready = FALSE;
15
```

```
16      TRANSITION
17      [
18        NOT (ready OR submitted) -->
19          ready' = TRUE;
20        []ready AND submitted -->
21          ready' = FALSE;
22      ];
23    END;
24
25    HDI: MODULE = system || % synchronously composed with EOFM handshake module
26      BEGIN
27        INPUT hLoosenOldControllerABCap: BOOLEAN
28        INPUT hTightenNewControllerABCap: BOOLEAN
29        INPUT hRedTagOldComponents: BOOLEAN
30        INPUT hSetAsideOldComponents: BOOLEAN
31        INPUT hRotateConnectorParts: BOOLEAN
32        INPUT hDisassembleConnector: BOOLEAN
33        INPUT hReassembleBrokenConnector: BOOLEAN
34        INPUT hDiscPumpCableFromAbCable: BOOLEAN
35        INPUT hDiscPumpCableFromOldController: BOOLEAN
36        INPUT hDiscLeadBattFromNewController: BOOLEAN
37        INPUT hDiscNewLiBattCableFromNewController: BOOLEAN
38        INPUT hConNewLiBattCableToNewController: BOOLEAN
39        INPUT hConLeadBattToNewController: BOOLEAN
40        INPUT hConPumpCableToNewController: BOOLEAN
41        INPUT hConNewLeadBattToNewController: BOOLEAN
42        INPUT hConNewLiBattCableToNewLiBatt: BOOLEAN
43        INPUT hDepressBlackButtonOnNewLiBatt: BOOLEAN
44        INPUT hRotateKnobClockwise: BOOLEAN
45        INPUT hRotateKnobCounterclockwise: BOOLEAN
46
47        INPUT ConnectorPartsRotatable: BOOLEAN
48        INPUT ConnectorPartsAssemblable: BOOLEAN
49        INPUT ConnectorPartsDisassemblable: BOOLEAN
50        INPUT NewLiBattCableDisconnectableFromNewController: BOOLEAN
51        INPUT LeadBattDisconnectableFromNewController: BOOLEAN
52        INPUT LeadBattConnectableToNewController: BOOLEAN
53        INPUT NewLiBattCableConnectableToNewLiBatt: BOOLEAN
54        INPUT NewLiBattCableConnectableToNewController: BOOLEAN
55        INPUT PumpCableConnectableToNewController: BOOLEAN
56        INPUT OldLiBattCableDisconnectableFromOldController: BOOLEAN
57        INPUT PumpCableDisconnectableFromOldController: BOOLEAN
58        INPUT PumpCableDisconnectableFromAbCable: BOOLEAN
59
60        OUTPUT iSpeedSetting: SpeedSettings
61        OUTPUT iOldComponentTags: userManual!tPartTag
62        OUTPUT iLeadBattToOldController: userManual!tConnection
63        OUTPUT iLeadBattToNewController: userManual!tConnection
64        OUTPUT iLeadBattToYCable: userManual!tConnection
65        OUTPUT iYCableToOldController: userManual!tConnection
66        OUTPUT iOldLiBattCableToYCable: userManual!tConnection
67        OUTPUT iOldLiBattCableToOldController: userManual!tConnection
68        OUTPUT iOldLiBattCableToOldLiBatt: userManual!tConnection
69        OUTPUT iPumpCableToOldController: userManual!tConnection
70        OUTPUT iAbCableToOldController: userManual!tConnection
71        OUTPUT iNewLiBattCableToNewLiBatt: userManual!tConnection
72        OUTPUT iNewLiBattCableToNewController: userManual!tConnection
73        OUTPUT iPumpCableToOldAbCable: userManual!tConnection
74        OUTPUT iPumpCableToNewController: userManual!tConnection
75        OUTPUT iNewLiBatteryLights: BatteryLights
76        OUTPUT iPermanentlyAttachedConnector: userManual!tPermAttachedConnectorStatus
77        OUTPUT iRotationCounter: userManual!tRotationCounter
78        OUTPUT iOldComponents: userManual!tOldComponentsLocation
79        OUTPUT iNewControllerABCap: userManual!tAlarmBatteryCap
80        OUTPUT iOldControllerABCap: userManual!tAlarmBatteryCap
81
82      INITIALIZATION
83        iPermanentlyAttachedConnector  IN {userManual!Broken, userManual!Assembled};
84        iSpeedSetting                  IN {1, 2, 3, 4, 5};
85        iOldComponents                 = userManual!AtHand;
86        iRotationCounter               = 0;
87        iNewLiBatteryLights            = 0;
88        iOldComponentTags              = userManual!notRedTagged;
89        iNewLiBattCableToNewLiBatt     = userManual!Disconnected;
90        iNewLiBattCableToNewController = userManual!Disconnected;
91        iLeadBattToNewController       = userManual!Disconnected;
92        iPumpCableToNewController      = userManual!Disconnected;
93        iOldControllerABCap  = userManual!Tightened;
94        iNewControllerABCap  = userManual!Loosened;
95        iYCableToOldController         IN {userManual!Connected, userManual!Disconnected};
96        iOldLiBattCableToOldController IN IF iYCableToOldController = userManual!Connected
```

```
97                                          THEN {userManual!Disconnected}
98                                          ELSE {userManual!Connected, userManual!Disconnected}
99                                        ENDIF;
100        iLeadBattToOldController       = IF iYCableToOldController = userManual!Connected
101                                          THEN userManual!Disconnected
102                                       ELSIF iOldLiBattCableToOldController = userManual!
               Connected
103                                          THEN userManual!Disconnected
104                                          ELSE userManual!Connected
105                                     ENDIF;
106        iLeadBattToYCable              = IF (iOldLiBattCableToYCable = userManual!Disconnected AND
               iYCableToOldController = userManual!Connected)
107                                          THEN userManual!Connected
108                                          ELSE userManual!Disconnected
109                                       ENDIF;
110        iOldLiBattCableToYCable        IN IF iYCableToOldController = userManual!Connected
111                                          THEN {userManual!Connected, userManual!Disconnected}
112                                          ELSE {userManual!Disconnected}
113                                        ENDIF;
114        iOldLiBattCableToOldLiBatt     = IF iOldLiBattCableToOldController = userManual!Connected
               OR (iYCableToOldController = userManual!Connected AND not(iLeadBattToYCable =
               userManual!Connected))
115                                          THEN userManual!Connected
116                                          ELSE userManual!Disconnected
117                                       ENDIF;
118
119        iPumpCableToOldAbCable         IN {userManual!Connected, userManual!Disconnected};
120        iPumpCableToOldController      = IF iPumpCableToOldAbCable = userManual!Connected
121                                          THEN userManual!Disconnected
122                                          ELSE userManual!Connected
123                                       ENDIF;
124        iAbCableToOldController        = IF iPumpCableToOldController = userManual!Connected
125                                          THEN userManual!Disconnected
126                                          ELSE userManual!Connected
127                                       ENDIF;
128        TRANSITION
129          [
130          hSetAsideOldComponents -->
131            iOldComponents' = userManual!SetAside;
132          []hRotateConnectorParts -->
133              iRotationCounter' = IF ConnectorPartsRotatable THEN iRotationCounter + 1 ELSE
                     iRotationCounter ENDIF;
134          []hReassembleBrokenConnector -->
135              iPermanentlyAttachedConnector' = IF ConnectorPartsAssemblable THEN userManual!
                     Assembled ELSE iPermanentlyAttachedConnector ENDIF;
136          []hDisassembleConnector -->
137              iPermanentlyAttachedConnector' = IF ConnectorPartsDisassemblable THEN userManual!
                     Broken ELSE iPermanentlyAttachedConnector ENDIF;
138          []hDiscPumpCableFromAbCable -->
139              iPumpCableToOldAbCable' = IF PumpCableDisconnectableFromAbCable THEN userManual!
                     Disconnected ELSE iPumpCableToOldAbCable ENDIF;
140          []hDiscPumpCableFromOldController -->
141              iPumpCableToOldController' = IF PumpCableDisconnectableFromOldController THEN
                     userManual!Disconnected ELSE iPumpCableToOldController ENDIF;
142          []hConNewLiBattCableToNewController -->
143              iNewLiBattCableToNewController' = IF NewLiBattCableConnectableToNewController THEN
                     userManual!Connected ELSE iNewLiBattCableToNewController ENDIF;
144          []hConNewLiBattCableToNewLiBatt -->
145              iNewLiBattCableToNewLiBatt' = IF NewLiBattCableConnectableToNewLiBatt THEN userManual
                     !Connected ELSE iNewLiBattCableToNewLiBatt ENDIF;
146          []hConLeadBattToNewController -->
147              iLeadBattToNewController' = IF LeadBattConnectableToNewController THEN userManual!
                     Connected ELSE iLeadBattToNewController ENDIF;
148          []hDiscLeadBattFromNewController -->
149              iLeadBattToNewController' = IF LeadBattDisconnectableFromNewController THEN
                     userManual!Disconnected ELSE iLeadBattToNewController ENDIF;
150          []hConPumpCableToNewController -->
151              iPumpCableToNewController' = IF PumpCableConnectableToNewController THEN userManual!
                     Connected ELSE iPumpCableToNewController ENDIF;
152          []hDiscNewLiBattCableFromNewController -->
153              iNewLiBattCableToNewController' = IF NewLiBattCableDisconnectableFromNewController
                     THEN userManual!Disconnected ELSE iNewLiBattCableToNewController ENDIF;
154          []hDepressBlackButtonOnNewLiBatt -->
155              iNewLiBatteryLights' IN {0, 1, 2, 3, 4, 5};
156          []hRotateKnobClockwise -->
157              iSpeedSetting' = iSpeedSetting - 1;
158          []hRotateKnobCounterclockwise -->
159              iSpeedSetting' = iSpeedSetting + 1;
160          []hRedTagOldComponents -->
161              iOldComponentTags' = userManual!redTagged;
162          []hLoosenOldControllerABCap -->
```

```
163             iOldControllerABCap' = userManual!Loosened;
164         []hTightenNewControllerABCap -->
165             iNewControllerABCap' = userManual!Tightened;
166         []ELSE -->
167       ];
168     END;
169     powerLightFunction(x: REAL): INTEGER =
170       IF      x <= 0.05                THEN 0
171         ELSIF x >  0.05 AND x <= 0.15 THEN 1
172         ELSIF x >  0.15 AND x <= 0.30 THEN 2
173         ELSIF x >  0.30 AND x <= 0.45 THEN 3
174         ELSIF x >  0.45 AND x <= 0.70 THEN 4
175         ELSIF x >  0.70 AND x <= 1.00 THEN 5
176         ELSIF x >  1.00 AND x <= 1.40 THEN 6
177         ELSIF x >  1.40 AND x <= 1.90 THEN 7
178         ELSIF x >  1.90 AND x <= 2.50 THEN 8
179         ELSIF x >  2.50 AND x <= 3.10 THEN 9
180         ELSIF x >  3.10 AND x <= 3.50 THEN 10
181       ELSE 11 ENDIF;
182
183   displayControl: MODULE =
184   BEGIN
185     INPUT iLeadBattToOldController: userManual!tConnection
186     INPUT iLeadBattToNewController: userManual!tConnection
187     INPUT iLeadBattToYCable: userManual!tConnection
188     INPUT iYCableToOldController: userManual!tConnection
189     INPUT iOldLiBattCableToYCable: userManual!tConnection
190     INPUT iOldLiBattCableToOldController: userManual!tConnection
191     INPUT iOldLiBattCableToOldLiBatt: userManual!tConnection
192     INPUT iPumpCableToOldController: userManual!tConnection
193     INPUT iAbCableToOldController: userManual!tConnection
194     INPUT iNewLiBattCableToNewLiBatt: userManual!tConnection
195     INPUT iNewLiBattCableToNewController: userManual!tConnection
196     INPUT iPumpCableToOldAbCable: userManual!tConnection
197     INPUT iPumpCableToNewController: userManual!tConnection
198     INPUT iPermanentlyAttachedConnector: userManual!tPermAttachedConnectorStatus
199     INPUT iNewControllerABCap: userManual!tAlarmBatteryCap
200     INPUT iOldControllerABCap: userManual!tAlarmBatteryCap
201     INPUT  speed: simData!speedRange
202     INPUT  power: simData!powerRange
203     OUTPUT functional: BOOLEAN
204     OUTPUT iPowerLight: PowerLights
205     OUTPUT iAlarm: Alarms
206     DEFINITION
207       functional IN
208         IF ((iPumpCableToOldAbCable = userManual!Connected AND iAbCableToOldController =
              userManual!Connected) OR
209            (iPumpCableToOldController = userManual!Connected)) AND
210              ((iLeadBattToYCable  = userManual!Connected AND iYCableToOldController =
                   userManual!Connected) OR
211              (iOldLiBattCableToYCable = userManual!Connected AND iYCableToOldController =
                   userManual!Connected) OR
212              (iLeadBattToOldController = userManual!Connected) OR
213              (iOldLiBattCableToOldController = userManual!Connected AND
                   iOldLiBattCableToOldLiBatt = userManual!Connected) OR
214              (iOldLiBattCableToYCable= userManual!Connected AND iYCableToOldController =
                   userManual!Connected AND iOldLiBattCableToOldLiBatt = userManual!Connected))
                    AND
215            iPermanentlyAttachedConnector = userManual!Assembled
216           THEN {TRUE, FALSE}
217         ELSIF
218              iPumpCableToNewController = userManual!Connected AND
219              iPermanentlyAttachedConnector = userManual!Assembled AND
220             (iLeadBattToNewController = userManual!Connected OR
221             (iNewLiBattCableToNewLiBatt = userManual!Connected AND iNewLiBattCableToNewController
                  = userManual!Connected))
222           THEN {TRUE}
223         ELSE {FALSE} ENDIF;
224
225       INITIALIZATION
226         [
227         speed = 0 -->
228           iAlarm = PumpStopped;
229           iPowerLight = powerLightFunction(power);
230         []speed > 0 AND power > 3.5 -->
231           iAlarm = HighPower;
232           iPowerLight = powerLightFunction(power);
233         []speed > 0 AND power <= 3.5 -->
234           iPowerLight = powerLightFunction(power);
235           iAlarm = NoAlarm;
236         []ELSE -->
```

```
237        ];
238        TRANSITION
239          [
240          speed' = 0 -->
241            iAlarm' = PumpStopped;
242            iPowerLight' = powerLightFunction(power');
243          []speed' > 0 AND power' > 3.5 -->
244            iAlarm' = HighPower;
245            iPowerLight' = powerLightFunction(power');
246          []speed' > 0 AND power' <= 3.5 -->
247            iAlarm' = NoAlarm;
248            iPowerLight' = powerLightFunction(power');
249          []ELSE -->
250        ];
251      END;
252  END
```

H.2.6 Continuous Device Model

```
1   simData: CONTEXT =
2     BEGIN
3     flowRange : TYPE = {x: REAL | x >= 0 AND x <= 7.25};
4     powerRange: TYPE = {x: REAL | x >= 0 AND x <= 5};
5     speedRange: TYPE = {x: REAL | x >= 0 AND x <= 6000};
6     two_k_RPM(flow: flowRange, speed: speedRange): [powerRange -> BOOLEAN];
7     three_k_RPM(flow: flowRange, speed: speedRange): [powerRange -> BOOLEAN];
8     four_k_RPM(flow: flowRange, speed: speedRange): [powerRange -> BOOLEAN];
9     five_k_RPM(flow: flowRange, speed: speedRange): [powerRange -> BOOLEAN];
10    six_k_RPM(flow: flowRange, speed: speedRange): [powerRange -> BOOLEAN];
11
12      slice1(inVars: [REAL, REAL]): BOOLEAN =
13        inVars = (1.000000,0.132929) OR
14        inVars = (1.250000,0.126067) OR
15        inVars = (1.500000,0.118702) OR
16        inVars = (1.750000,0.110974) OR
17        inVars = (2.000000,0.101147) OR
18        inVars = (2.250000,0.090146) OR
19        inVars = (2.500000,0.078316) OR
20        inVars = (0.000000,0.000000);
21      slice2(inVars: [REAL, REAL]): BOOLEAN =
22        inVars = (1.000000,0.455320) OR
23        inVars = (1.250000,0.444335) OR
24        inVars = (1.500000,0.430959) OR
25        inVars = (1.750000,0.413479) OR
26        inVars = (2.000000,0.393657) OR
27        inVars = (2.250000,0.374802) OR
28        inVars = (2.500000,0.355139) OR
29        inVars = (2.750000,0.333257) OR
30        inVars = (3.000000,0.306309) OR
31        inVars = (3.250000,0.280449) OR
32        inVars = (3.500000,0.252173) OR
33        inVars = (3.750000,0.219310) OR
34        inVars = (4.000000,0.184870) OR
35        inVars = (4.250000,0.146359) OR
36        inVars = (0.000000,0.000000);
37      slice3(inVars: [REAL, REAL]): BOOLEAN =
38        inVars = (1.000000,1.077672) OR
39        inVars = (1.250000,1.057788) OR
40        inVars = (1.500000,1.037536) OR
41        inVars = (1.750000,1.016654) OR
42        inVars = (2.000000,0.992567) OR
43        inVars = (2.250000,0.961997) OR
44        inVars = (2.500000,0.924922) OR
45        inVars = (2.750000,0.887002) OR
46        inVars = (3.000000,0.849523) OR
47        inVars = (3.250000,0.815588) OR
48        inVars = (3.500000,0.776701) OR
49        inVars = (3.750000,0.725160) OR
50        inVars = (4.000000,0.674518) OR
51        inVars = (4.250000,0.626903) OR
52        inVars = (4.500000,0.571348) OR
53        inVars = (4.750000,0.512964) OR
54        inVars = (5.000000,0.453164) OR
55        inVars = (5.250000,0.393815) OR
56        inVars = (5.500000,0.327378) OR
57        inVars = (5.750000,0.255880) OR
58        inVars = (6.000000,0.176579) OR
59        inVars = (0.000000,0.000000);
60      slice4(inVars: [REAL, REAL]): BOOLEAN =
61        inVars = (1.000000,2.087428) OR
```

```
62          inVars = (1.250000,2.060755) OR
63          inVars = (1.500000,2.031205) OR
64          inVars = (1.750000,1.998279) OR
65          inVars = (2.000000,1.963199) OR
66          inVars = (2.250000,1.930934) OR
67          inVars = (2.500000,1.895294) OR
68          inVars = (2.750000,1.851050) OR
69          inVars = (3.000000,1.789014) OR
70          inVars = (3.250000,1.727843) OR
71          inVars = (3.500000,1.663063) OR
72          inVars = (3.750000,1.605113) OR
73          inVars = (4.000000,1.551224) OR
74          inVars = (4.250000,1.488895) OR
75          inVars = (4.500000,1.412560) OR
76          inVars = (4.750000,1.322547) OR
77          inVars = (5.000000,1.251706) OR
78          inVars = (5.250000,1.168460) OR
79          inVars = (5.500000,1.076372) OR
80          inVars = (5.750000,0.984535) OR
81          inVars = (6.000000,0.890290) OR
82          inVars = (0.000000,0.000000);
83        slice5(inVars: [REAL, REAL]): BOOLEAN =
84          inVars = (1.000000,3.570688) OR
85          inVars = (1.250000,3.534916) OR
86          inVars = (1.500000,3.496901) OR
87          inVars = (1.750000,3.456682) OR
88          inVars = (2.000000,3.411798) OR
89          inVars = (2.250000,3.358940) OR
90          inVars = (2.500000,3.306075) OR
91          inVars = (2.750000,3.263566) OR
92          inVars = (3.000000,3.215792) OR
93          inVars = (3.250000,3.147522) OR
94          inVars = (3.500000,3.064465) OR
95          inVars = (3.750000,2.964712) OR
96          inVars = (4.000000,2.877546) OR
97          inVars = (4.250000,2.782650) OR
98          inVars = (4.500000,2.700965) OR
99          inVars = (4.750000,2.620880) OR
100         inVars = (5.000000,2.529830) OR
101         inVars = (5.250000,2.420061) OR
102         inVars = (5.500000,2.292861) OR
103         inVars = (5.750000,2.184787) OR
104         inVars = (6.000000,2.075238) OR
105         inVars = (6.250000,1.938593) OR
106         inVars = (6.500000,1.799692) OR
107         inVars = (6.750000,1.661903) OR
108         inVars = (7.000000,1.532875) OR
109         inVars = (7.250000,1.398106) OR
110         inVars = (0.000000,0.000000);
111     simData_Constraints: MODULE =
112     BEGIN
113       INPUT flow:  flowRange
114       INPUT speed: speedRange
115       INPUT power: powerRange
116       LOCAL var_1: REAL
117       LOCAL var_2: REAL
118       OUTPUT fitsData: BOOLEAN
119       DEFINITION
120         var_1 = flow;
121         var_2 = power;
122       INITIALIZATION [
123           slice1(var_1,var_2) AND speed = 2000 -->
124           fitsData = true;
125         []slice2(var_1,var_2) AND speed = 3000 -->
126           fitsData = true;
127         []slice3(var_1,var_2) AND speed = 4000 -->
128           fitsData = true;
129         []slice4(var_1,var_2) AND speed = 5000 -->
130           fitsData = true;
131         []slice5(var_1,var_2) AND speed = 6000 -->
132           fitsData = true;
133         []var_1 = 0 AND var_2 = 0 AND speed = 0 -->
134           fitsData = true;
135         []ELSE -->
136           fitsData = false;
137       ];
138
139       TRANSITION [
140           not(slice1(var_1',var_2')) AND speed' = 2000 -->
141           fitsData' = false;
142         []not(slice2(var_1',var_2')) AND speed' = 3000 -->
```

```
143          fitsData' = false;
144        []not(slice3(var_1',var_2')) AND speed' = 4000 -->
145          fitsData' = false;
146        []not(slice4(var_1',var_2')) AND speed' = 5000 -->
147          fitsData' = false;
148        []not(slice5(var_1',var_2')) AND speed' = 6000 -->
149          fitsData' = false;
150        []not(var_1 = 0 AND var_2 = 0 AND speed = 0) -->
151          fitsData' = false;
152        []ELSE -->
153      ];
154
155    END;
156
157    device: MODULE =
158    BEGIN
159      INPUT  iSpeedSetting: NATURAL
160      INPUT  functional: BOOLEAN
161      OUTPUT flow:  flowRange
162      OUTPUT speed: speedRange
163      OUTPUT power: powerRange
164    INITIALIZATION
165        [
166         iSpeedSetting = 1 AND functional -->
167          speed =  2000;
168          power =  0.132929;
169          flow = 1;
170        []iSpeedSetting = 2 AND functional -->
171          speed =  3000;
172          power =  0.455320;
173          flow = 1;
174        []iSpeedSetting = 3 AND functional -->
175          speed =  4000;
176          power =  1.077672;
177          flow = 1;
178        []iSpeedSetting = 4 AND functional -->
179          speed =  5000;
180          power =  2.087428;
181          flow = 1;
182        []iSpeedSetting = 5 AND functional -->
183          speed =  6000;
184          power =  3.570688;
185          flow = 1;
186        []not(functional) -->
187          speed =  0;
188          power =  0;
189          flow = 0;
190        []ELSE -->
191      ];
192    TRANSITION
193      [ iSpeedSetting' = 1 AND functional' -->
194          speed' = 2000;
195          flow'  IN {1, 1.25, 1.5, 1.75, 2, 2.25, 2.5};
196          power' IN two_k_RPM(flow', speed');
197      []iSpeedSetting' = 2 AND functional' -->
198          speed' = 3000;
199          flow'  IN {1, 1.25, 1.5, 1.75, 2, 2.25, 2.5, 2.75, 3, 3.25, 3.5, 3.75, 4};
200          power' IN three_k_RPM(flow', speed');
201      []iSpeedSetting' = 3 AND functional' -->
202          speed' = 4000;
203          flow'  IN {1, 1.25, 1.5, 1.75, 2, 2.25, 2.5, 2.75, 3, 3.25, 3.5,
204                 3.75, 4, 4.25, 4.5, 4.75, 5, 5.25, 5.5, 5.75, 6};
205          power' IN four_k_RPM(flow', speed');
206      []iSpeedSetting' = 4 AND functional' -->
207          speed' = 5000;
208          flow'  IN {1, 1.25, 1.5, 1.75, 2, 2.25, 2.5, 2.75, 3, 3.25, 3.5,
209                 3.75, 4, 4.25, 4.5, 4.75, 5, 5.25, 5.5, 5.75, 6};
210          power' IN five_k_RPM(flow', speed');
211      []iSpeedSetting' = 5 AND functional' -->
212          speed' = 6000;
213          flow'  IN {1, 1.25, 1.5, 1.75, 2, 2.25, 2.5, 2.75, 3, 3.25, 3.5,
214                 3.75, 4, 4.25, 4.5, 4.75, 5, 5.25, 5.5, 5.75, 6, 6.25,
215                 6.5, 6.75, 7, 7.25};
216          power' IN six_k_RPM(flow', speed');
217      []not(functional') -->
218          speed' = 0;
219          power' = 0;
220          flow' = 0;
221      []ELSE -->
222    ];
223    END;
```

```
224    actuator: MODULE = simData_Constraints || device;
225  END
```

H.3 Specifications

H.3.1 Accuracy and Understandability

```
1  G(fitsData AND iAlarm = discreteDevice!PumpStopped AND aRespondToPumpStoppedAlarm_Executing =>
2        PumpStoppedAlarm.Color.PumpSpeed = SpeedSettingKnob.Color.PumpSpeed
3    AND SpeedSettingKnob.Color.PumpSpeed = SpeedSettingKnob.Label.PumpSpeed
4    AND SpeedSettingKnob.Label.PumpSpeed = Stopped
5    AND Audibly_Signified_PumpSpeed = Visually_Signified_PumpSpeed);
```

H.3.2 Accuracy and Error Tolerance

```
1  G(fitsData AND aRespondToPumpStoppedAlarm_Executing
2    AND (iOldLiBattCableToOldLiBatt = Connected OR iOldLiBattCableToOldController = Connected) =>
3      NOT(OldLiBattCableConnectableToNewController OR NewLiBattCableConnectableToOldLiBatt));
```

H.3.3 Accuracy and Time Efficiency

```
1  G(fitsData AND aRespondToPumpStoppedAlarm_Executing AND iNewLiBattLights = 0
2    AND X(fitsData AND iNewLiBattLights = 5) =>
3        X(NewLiBattCableConnectableToNewController
4        AND NewLiBattCableConnectableToNewLiBatt));
```

H.3.4 Accuracy and Completeness

```
1  G(fitsData AND aAdjustSpeed_Executing AND iAlarm = discreteDevice!NoAlarm =>
2      Visually_Signified_PumpSpeed = Stopped
3      OR Audibly_Signified_PumpSpeed = Stopped
4      OR Documented_PumpSpeed = Stopped));
```

H.3.5 Understandability and Error Tolerance

```
1  G(iOldLiBattCableToOldLiBatt = userManual!Connected =>
2    NOT OldLiBattCableConnectableToNewController)
3    AND G(fitsData =>
4      AND PumpStoppedAlarm.Color.PumpSpeed = SpeedSettingKnob.Color.PumpSpeed
5      AND SpeedSettingKnob.Color.PumpSpeed = SpeedSettingKnob.Label.PumpSpeed
6      AND PowerIndicators.Color.PowerSupplied = PowerIndicators.Label.PowerSupplied
7      AND PowerIndicators.Label.PowerSupplied = PowerIndicators.Color.PowerSupplied
8      AND PumpStoppedAlarm.Volume.PumpSpeed = PumpStoppedAlarm.aPattern.PumpSpeed
9      AND PowerIndicators.Volume.PowerSupplied = PowerIndicators.aPattern.PowerSupplied);
```

H.3.6 Understandability and Time Efficiency

```
1  G(fitsData AND iAlarm = discreteDevice!PumpStopped
2    AND X(fitsData AND iAlarm = discreteDevice!HighPower) =>
3        X(PumpStoppedAlarm.Color.PumpSpeed = SpeedSettingKnob.Color.PumpSpeed
4      AND SpeedSettingKnob.Color.PumpSpeed = SpeedSettingKnob.Label.PumpSpeed
5      AND PowerIndicators.Color.PowerSupplied = PowerIndicators.Label.PowerSupplied
6      AND PowerIndicators.Label.PowerSupplied = PowerIndicators.Color.PowerSupplied
7      AND PumpStoppedAlarm.Volume.PumpSpeed = PumpStoppedAlarm.aPattern.PumpSpeed
8      AND PowerIndicators.Volume.PowerSupplied = PowerIndicators.aPattern.PowerSupplied
9          Audibly_Signified_PowerSupplied = Visually_Signified_PowerSupplied));
```

H.3.7 Error Tolerance and Time Efficiency

```
1  G(iOldLiBattCableToOldController = Connected
2    AND iNewLiBattLights = 0 AND X(iNewLiBattLights = 5) =>
3        X(NOT NewLiBattCableConnectableToOldLiBatt
4          AND NewLiBattCableConnectableToNewLiBatt));
```

H.3.8 Error Tolerance and Completeness

```
1  G(iOldLiBattCableToOldController = userManual!Connected =>
2      NOT OldLiBattConnectableToNewLiBattCable)
3    AND G(fitsData =>
4      NOT(Visually_Signified_PumpSpeed = bigsis!PumpSpeedNotSignified
5        AND Audibly_Signified_PumpSpeed = bigsis!PumpSpeedNotSignified
6        AND Documented_PumpSpeed = bigsis!PumpSpeedNotSignified
```

```
7        AND Visually_Signified_PowerSupplied = bigsis!PowerSuppliedNotSignified
8        AND Audibly_Signified_PowerSupplied = bigsis!PowerSuppliedNotSignified
9        AND Documented_PowerSupplied = bigsis!PowerSuppliedNotSignified));
```

H.3.9 Understandability and Completeness

```
1  G(fitsData =>
2        NOT(Visually_Signified_PumpSpeed = bigsis!PumpSpeedNotSignified AND
3            Audibly_Signified_PumpSpeed = bigsis!PumpSpeedNotSignified AND
4            Documented_PumpSpeed = bigsis!PumpSpeedNotSignified)
5    AND NOT(Visually_Signified_PowerSupplied = bigsis!PowerSuppliedNotSignified AND
6            Audibly_Signified_PowerSupplied = bigsis!PowerSuppliedNotSignified AND
7            Documented_PowerSupplied = bigsis!PowerSuppliedNotSignified)
8    AND (PumpStoppedAlarm.Color.PumpSpeed = SpeedSettingKnob.Color.PumpSpeed AND
9         SpeedSettingKnob.Color.PumpSpeed = SpeedSettingKnob.Label.PumpSpeed AND
10        PowerIndicators.Color.PowerSupplied = PowerIndicators.Label.PowerSupplied AND
11        PowerIndicators.Label.PowerSupplied = PowerIndicators.Color.PowerSupplied)
12   AND (PumpStoppedAlarm.Volume.PumpSpeed = PumpStoppedAlarm.aPattern.PumpSpeed AND
13        bPowerIndicators.Volume.PowerSupplied = PowerIndicators.aPattern.PowerSupplied));
```

H.3.10 Time Efficiency and Completeness

```
1  G(fitsData AND iSpeedSetting /= lDesiredSpeed
2    AND X(fitsData AND iSpeedSetting = lDesiredSpeed) =>
3      X(NOT(Visually_Signified_PumpSpeed = bigsis!PumpSpeedNotSignified
4        AND Audibly_Signified_PumpSpeed = bigsis!PumpSpeedNotSignified
5        AND Documented_PumpSpeed = bigsis!PumpSpeedNotSignified
6        AND Visually_Signified_PowerSupplied = bigsis!PowerSuppliedNotSignified
7        AND Audibly_Signified_PowerSupplied = bigsis!PowerSuppliedNotSignified
8        AND Documented_PowerSupplied = bigsis!PowerSuppliedNotSignified)));
```

Vita

Andrew J. Abbate received the B.A. degree in economics from Boston College in 2013, and the M.S. degree in biomedical science from Drexel University in 2015. His research interests include the application of formal methods in the design and usability evaluation of safety-critical systems, with concentrations in medical devices and air transportation systems.

His Ph.D. research at Drexel University was conducted in the BioCirc Research Laboratory (Director: Amy L. Throckmorton, Ph.D.) and the Human-Systems Evaluation and Analysis Laboratory (Director: Ellen J. Bass, Ph.D.), and his expected graduation date is March 2017. As a teaching assistant at Drexel, he held weekly office hours and graded homework assignments for BMES-505–507, Math for Biomedical Scientists I–III. He also delivered a guest lecture for the Graduate Seminar in Health Informatics at Temple University in Philadelphia, PA.

His honors and awards include:

- 2016 Human Factors and Ergonomics Society (HFES) Student Member With Honors
- 2015–2017 U.S. Department of Education Graduate Assistance in Areas of National Need (GAANN) fellow
- 2016 Human-Computer Interaction Consortium (HCIC) fully funded student attendee
- 2015 HFES International Meeting, 2nd place winner for best student paper submitted to the Safety Technical Group
- 2015 HFES International Meeting, recipient of a travel award from the Drexel University Graduate College for presenting work

His archival journal articles and refereed conference proceedings/abstracts include:

1. A.J. Abbate, E.J. Bass, and A.L. Throckmorton. (2016). "A formal task analytic approach to medical device alarm troubleshooting instructions." *Human-Machine Systems, IEEE Transactions on*, 41(1), pp. 53–65.
2. A.J. Abbate and E.J. Bass. "A formal language for specifying visual interface signifiers," in *Proceedings of the 2016 Annual Meeting of the Human Factors and Ergonomics Society*, Washington D.C., USA, September 19–23, 2016. pp, 1098-1102.
3. A.J. Abbate and E.J. Bass. "Using computational tree logic methods to analyze reachability in user documentation," in *Proceedings of the 2015 Annual Meeting of the Human Factors and Ergonomics Society*, Los Angeles, CA, USA, October 26–30, 2015, pp. 1481–1485.
4. A.J. Abbate, E.J. Bass, and A.L. Throckmorton. "Using formal task analytic models to support user manual development: An LVAD case study," in *Proceedings of the International Symposium on Human Factors and Ergonomics in Health Care*, Baltimore, MD, USA, April 26–29, 2015, pp. 114–117.
5. A.J. Abbate, E.J. Bass, and A.L. Throckmorton. "Formal Usability Methods to Support the Expedited Access Paradigm: A Pediatric Blood Pump Case Study." Abstract presented at ASAIO 62nd Annual Conference in San Francisco, CA, USA, June 15–18, 2016.