View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by MURAL - Maynooth University Research Archive Library

Event-B in the Institutional Framework:

Defining a Semantics, Modularisation Constructs and Interoperability for a Specification Language

MARIE FARRELL

A thesis submitted for the degree of

Doctor of Philosophy

Department of Computer Science

Maynooth University

October 2017

Supervisors: Dr. Rosemary Monahan & Dr. James F. Power

Head of Department: Prof. Adam Winstanley

CONTENTS

0 INTRODUCTION AND BACKGROUND MATERIAL 0
1 INTRODUCTION 1
1.1 Motivation . . . 3
1.2 Thesis Statement . . . 4
1.3 Summary of Contributions . . . 5
2 BACKGROUND MATERIAL 8
2.1 Event-B . . . 8
2.1.1 Example: Traffic-Lights System . . . 10
2.1.2 Tool Support and Proof . . . 12
2.2 Limitations of Event-B . . . 13
2.2.1 Modularisation as Decomposition in Event-B . . . 14
2.2.2 Interoperability and Heterogeneity . . . 20
2.3 The Theory of Institutions . . . 23
2.3.1 Category-Theoretic Prerequisites . . . 24
2.3.2 Institutions . . . 27
2.3.3 Example: $\mathcal{FOPEQ}$ - the Institution for First-Order Predicate Logic with Equality . . . 28
2.3.4 Refinement . . . 29
2.4 Addressing the Limitations of Event-B . . . 31
2.4.1 Institution-Theoretic Modularisation Constructs . . . 31
2.4.2 Institution-Theoretic Interoperability . . . 34
2.5 Summary . . . 39
I DEFINING A SEMANTICS 40
3 DEFINING $\mathcal{EVT}$ - AN INSTITUTION FOR EVENT-B 41
3.1 Introducing $\mathcal{EVT}$ . . . 41

3.2 $\mathbf{Sign}_{\mathcal{EVT}}$, the Category of $\mathcal{EVT}$-signatures 42
3.3 The Functor $\mathbf{Sen}_{\mathcal{EVT}}$, yielding $\mathcal{EVT}$-sentences 45
3.4 The Functor $\mathbf{Mod}_{\mathcal{EVT}}$, yielding $\mathcal{EVT}$-models 50
3.5 The Satisfaction Relation for $\mathcal{EVT}$ 56
3.6 Relating $\mathcal{FOPEQ}$ and $\mathcal{EVT}$ 59
3.7 Pushouts and Amalgamation 62
3.8 Pragmatics of Specification Building in $\mathcal{EVT}$ 68
3.9 Writing Specifications in the $\mathcal{EVT}$ Institution 69
3.9.1 Representing Refinement Explicitly 73
3.10 Summary 74
4 FORMALISING A TRANSLATIONAL SEMANTICS FOR EVENT-B 75
4.1 Introduction 75
4.2 Syntax of Event-B 76
4.3 A $\mathcal{FOPEQ}$ Interface 79
4.4 Extracting the Signature 80
4.5 Defining the Semantics of Event-B Superstructure Sentences 83
4.6 Defining the Semantics of Event-B Infrastructure Sentences 90
4.7 Implementing the Translational Semantics 92
4.8 Applying the Translational Semantics to an Example . . . 93
4.8.1 The Abstract Model 93
4.8.2 The First Refinement 95
4.8.3 The Second Refinement 95
4.9 Summary 100
II INTEROPERABILITY 101
5 AN INSTITUTION-THEORETIC FOUNDATION FOR THE TRANSLATION FROM UML TO EVENT-B 102
5.1 Introduction 102
5.2 $\mathcal{UML}$ - The Institution for Simple UML State Machines . . 103
5.2.1 $\mathcal{ACT}$ - The Underlying Action Institution 104
5.2.2 The Behavioural State Machine Institution 108

5.2.3 The Protocol State Machine Institution 110
5.2.4 The State Machine Tripod of Institutions 111
5.3 Translating from $\mathcal{UML}$ to $\mathcal{EVT}$ via an Institution Comorphism 111
5.3.1 Comparing $\mathcal{EVT}$ and $\mathcal{UML}$ 111
5.3.2 Relating the Action Institution and $\mathcal{FOPEQ}$ 113
5.3.3 Relating the $\mathcal{UML}$ Institution and $\mathcal{EVT}$ 113
5.4 Example 120
5.4.1 Preservation of the Satisfaction Condition 124
5.4.2 Analysing a Selection of Potential Edge Cases . . . 125
5.5 The Comorphism as a Foundation for UML-B 126
5.5.1 Refinement 129
5.6 Summary 134
III MODULARISATION 136
6 SPECIFICATION CLONES: AN EMPIRICAL STUDY OF THE STRUCTURE OF EVENT-B SPECIFICATIONS 137
6.1 Background and Introduction 137
6.1.1 Clones in Code and Specifications 138
6.1.2 Modularisation of Event-B Specifications 139
6.2 Analysing a Corpus of Event-B specifications: Metrics and Refinement 139
6.2.1 Quantifying Specification Size 142
6.2.2 Metrics for Event-B Specifications 144
6.2.3 Quantifying Refinements 146
6.3 Detecting Specification Clones 147
6.4 Results of the Clone Analysis 150
6.4.1 Context Clones 150
6.4.2 Machine Clones 151
6.4.3 Event Clones 152
6.5 Decloning Event-B Specifications 153

6.5.1 Decloning Contexts 155
6.5.2 Decloning Machines 155
6.5.3 Decloning Events 156
6.6 Threats to Validity 157
6.7 Summary and Future Work 158

IV CONCLUSIONS 160

7 CONCLUSIONS AND FUTURE WORK 161
7.1 Future Work 161
7.2 Summary 162

A DECLONING EVENT-B SPECIFICATIONS USING SPECIFICATION-BUILDING OPERATORS 165
A.1 Rodin plugins 165
A.1.1 Feature Composition 165
A.1.2 Generic Instantiation 167
A.1.3 Model Decomposition 169
A.1.4 Pattern 170
A.1.5 Shared Event (Parallel) Composition 172
A.1.6 Modularisation Plugin 173
A.1.7 XEvent-B 174
A.1.8 Related Plugins 174
A.1.9 Rodin Compatabilities 176
A.2 Refinement as a Modularisation Technique 176
A.3 Discussion 179

B MATHEMATICAL NOTATION AND SOFTWARE ARTEFACTS 181
B.1 Mathematical Notation 181
B.2 Institutions 181
B.3 Institution Comorphisms and Semi-Morphisms 182
B.4 Software Artefacts 182
B.4.1 clonedetector 183
B.4.2 EB2EVT 184

B.4.3 EVTHets 184
B.4.4 Specs 185

BIBLIOGRAPHY 186

LIST OF FIGURES

Figure 1.1 Interoperability can be achieved between UML and Event-B using their respective institutions and our EB2EVT translator tool. 7

Figure 2.1 Event-B specification of a traffic system. Lines 1–21 contain an Event-B specification for an abstract machine that uses boolean flags to describe the behaviour of the traffic system. The Event-B context on lines 22–27 provides a specification for the colours of a set of traffic–lights. Lines 28–59 contain a refined Event-B machine specification for a traffic system. 10

Figure 2.2 The shared variable decomposition of machine M into submachines M1 and M2. 16

Figure 2.3 The shared event decomposition of machine M into submachines M1 and M2. 16

Figure 2.4 The modularisation approach to decomposition using module interfaces and event groups. 18

Figure 3.1 The elements of an Event-B machine as presented in Rodin and their corresponding $\mathcal{EVT}$-sentences. . 46

Figure 3.2 These are the $\mathcal{EVT}$-sentences corresponding to the abstract Event-B traffic light system as illustrated on lines 1–21 of Figure 2.1. 48

Figure 3.3 An example of an Event-B event, e, with natural number variable x and boolean variable y. When $x > 2$, the event increments the value of x and toggles y to false. 51

Figure 3.4 The construction of $R' = R_1 \otimes R_2$, the amalgamation of R_1 and R_2. 66
Figure 3.5 A modular institution-based presentation corresponding to the abstract machine mac1 in Fig 2.1. . 71
Figure 3.6 A modular institution-based presentation corresponding to the refined machine mac2 specified in Figure 2.1 (lines 28–59). 72
Figure 3.7 Defining the refinement relationships between the concrete and abstract presentations. 74
Figure 4.1 We split the Event-B syntax into three components: superstructure, infrastructure and a mathematical language 77
Figure 4.2 The Event-B syntax is parametrised by first-order logic as indicated by our use of the nonterminals *predicate* and *expression*. These will be mapped to $\mathcal{FOPEQ}$-formulae and terms respectively in our translational semantics. 78
Figure 4.3 The $\mathcal{FOPEQ}$ interface provides access to a range of operations and semantic functions which we assume to exist. These are used throughout our semantic definitions in Figures 4.4, 4.5, 4.7, 4.8, 4.9 and 4.10. 80
Figure 4.4 The semantics of $\mathcal{EVT}$-signature extraction for machines. 81
Figure 4.5 The semantics of $\mathcal{FOPEQ}$-signature extraction uses the interface described in Figure 4.3 in order to extract signature components from the definition of a *ContextBody*. 82

Figure 4.6 Signature extracted by application of the semantic functions in Figures 4.4 and 4.5 to the Event-B machine specification of `mac2` in Figure 2.1. 83

Figure 4.7 The semantics of Event-B superstructure sentences are defined by translating them into presentations over $\mathcal{EVT}$ using the semantic function $\mathbb{B}$ and the specification-building operators defined in the theory of institutions (Table 2.3). Recall from Definition 29 that objects of **Pres** are of the form $\langle \Sigma, \Phi \rangle$ for a signature Σ and $\Phi \subseteq Sen(\Sigma)$. This figure contains the translation for machine specifications, the translation of events and contexts is outlined in Figure 4.8. 85

Figure 4.8 The translation of the event and context components of the Event-B superstructure sentences. This is a continuation of the translation described in Figure 4.7. 86

Figure 4.9 A semantics for Event-B infrastructure sentences is provided by translating them into sentences over $\mathcal{EVT}$, denoted $Sen_{\mathcal{EVT}}(\Sigma)$, for machines and sentences over $\mathcal{FOPEQ}$, denoted $Sen_{\mathcal{FOPEQ}}(\Sigma)$, for contexts. We use the interface operations and semantic functions described in Figure 4.3 throughout this translation. The event and context components of this translation are contained in Figure 4.10. 87

Figure 4.10 This is a continuation of the Event-B infrastructure sentence translation outlined in Figure 4.9. Here, we provide the event and context specific components of the translation of the infrastructure sentences. 88
Figure 4.11 The Haskell implementation of the $\mathbb{B}$ function from Figure 4.9 as applied to a list of Event-B event definitions. 92
Figure 4.12 Event-B abstract machine `m0 cd`. This specification consists of the events `ML_out` and `ML_in` that model the behaviour of cars leaving and entering the mainland respectively. 93
Figure 4.13 Syntactically sugared $\mathcal{EVT}$-specification as generated by `EB2EVT` that corresponds to the Event-B model in Figure 4.12. 94
Figure 4.14 Event-B machine `m1` with additional events `IL_in` and `IL_out` to model the behaviour of cars entering and leaving the island. The variables `a`, `b`, and `c` keep track of the number of cars on the bridge going to the island, the number of cars on the mainland and the number of cars on the bridge going to the mainland respectively. 96
Figure 4.15 $\mathcal{EVT}$-specification corresponding to the first refinement step as presented in Figure 4.14. 97
Figure 4.16 Event-B machine `m2` that refines the Event-B machine in Figure 4.14 by adding new events `Ml_tl_green` and `Il_tl_green`. The context `Color` on lines 1–8 adds a new data type which is used by the `ml_tl` and `il_tl` traffic light variables. 98

Figure 4.17 $\mathcal{EVT}$-specification corresponding to the second refinement step as presented in Figure 4.16. 99
Figure 5.1 An overview of our approach to interoperability between UML and Event-B. 103
Figure 5.2 The state machine tripod of institutions, encompassing the institutions for behavioural and protocol state machines, is formed using an underlying institution of actions $\mathcal{ACT}$. 104
Figure 5.3 UML state machine describing an automated teller machine (ATM) based on the example in Knapp et al. [73]. 120
Figure 5.4 Preprocessed version of the state machine from Figure 5.3. Note that we have added the intermediate state `Inter1` during the preprocessing phase in order to split up the method calls `userCom.keepCard()` and `bankCom.markInvalid(cardId)`.121
Figure 5.5 UML-B representation of the ATM state machine described in Figure 5.4. Note the addition of the completion events `PINEntered`, `Inter1` and `Verified` as transitions. 127
Figure 5.6 These are the $\mathcal{EVT}$-sentences extracted from the Event-B specification shown in Figure 5.8. $\overline{x}$ and $\overline{x}\prime$ can be inferred from V, described above. 129
Figure 5.7 This is the context that was generated by the UML-B plugin. It specifies a new datatype called *behavioural_STATES* that allows us to refer to the states in the corresponding UML state machine. 129

Figure 5.8 The Event-B machine generated from the UML-B model shown in Figure 5.5. Notice that every transition in the state machine corresponds to an event in this Event-B machine. 130
Figure 5.9 Protocol state machine from [73] as represented using the UML-B plugin. 131
Figure 5.10 A refinement cube showing the various levels at which refinement takes place throughout the ATM example using the UML and Event-B formalisms and their respective institutions. 132
Figure 5.11 The Event-B specification that was generated for the abstract protocol machine that was shown in Figure 5.9. 133
Figure 5.12 This figure illustrates that the proof obligations associated with the Event-B models produced are discharged automatically. 134
Figure 6.1 Histograms showing the distribution of the numbers of sentences per project for the smaller and larger data sets. Note that the vertical axes here are on different scales. 144
Figure 6.2 Histograms with kernel distribution describing the number of refinement steps taken in both the smaller and larger project sets. Note that the vertical axes here are on different scales. 148
Figure 6.3 Histograms describing the distribution of Type-3 clones across the entire corpus of Event-B specifications. Note that we have omitted type-3 *context* clones as there were relatively few of these. 153

Figure 6.4 Decloning the context clone that appeared on an intra and inter project basis throughout the Hemodialysis Machine developments. 155
Figure 6.5 This figure illustrates how superposition refinement between an abstract and a concrete specification can be represented using the `then` specification-building operator. Here the two `Initialisation` events are merged as they have the same name. . 156
Figure 6.6 Decloning the events in the DepSatSpec Event-B specification using specification-building operators.157
Figure A.1 We use the `and` operator to completely combine features given by the machines `mac1` and `mac2` (lines 1–3) which have already been specified. We use `and` in combination with the `with` operator to ensure that the specifications of `mac1` and `mac2` are disjoint before they are merged (lines 4–7). We use the `hide via` σ operator to partially compose the features given by machines `mac1` and `mac2` (lines 8–17). 167
Figure A.2 An instantiated machine obtained using the generic instantiation plugin is shown on lines 1–11. It is represented using specification-building operators on lines 12–17. Lines 18–27 use parametrisation to describe the same machine. 169
Figure A.3 Writing the shared variable style used in Figure 2.2 using specification-building operators 170
Figure A.4 Incorporating a `pattern` machine in the current development `mac1` 171
Figure A.5 Shared Event Composition using the specification-building operators 172

Figure A.6 Representing the functionality of the XEvent-B plugin using specification-building operators. . . . 174
Figure A.7 A parametrised version of the simple traffic light system that was illustrated in Figure 2.1. 179

LIST OF TABLES

Table 2.1 Rodin plugins that employ modularisation features. 19
Table 2.2 Rodin plugins that support interoperability for Event-B. 21
Table 2.3 Institution-theoretic specification-building operators that can be used to modularise specifications in a formalism-independent manner. Note that $SP1$ and $SP2$ denote specifications written over some institution, and σ is a signature morphism in the same institution. 32
Table 5.1 Table of symbols summarising the components of the action institution, $\mathcal{ACT}$ (Section 5.2.1), the UML state machine institution, $\mathcal{UML}$ (Section 5.2.3) and the institution for Event-B, $\mathcal{EVT}$ (Chapter 3). . 105
Table 6.1 Metrics for the Event-B projects that fall into the "smaller" category. 140
Table 6.2 Metrics for the Event-B projects that fall into the "larger" category. Outliers are indicated by an asterisk*. 141
Table 6.3 Summary statistics for the whole data set, and for the two "smaller" and "larger" subdivisions. . . . 141

Table 6.4 The occurrence of clone pairs and clones per type throughout the entire corpus. Note that '(+VI)' indicates that the variants (where appropriate) and invariants have been included in the analysis. . . . 151
Table A.1 Table summarising the latest version of the *Rodin Platform* that each of the identified plugins is compatible with. 176
Table B.1 Summary of the mathematical notation used throughout this thesis. 182

ABSTRACT

Event-B is an industrial-strength specification language for verifying the properties of a given system's specification. It is supported by its Eclipse-based IDE, Rodin, and uses the process of refinement to model systems at different levels of abstraction. Although a mature formalism, Event-B has a number of limitations. In this thesis, we demonstrate that Event-B lacks formally defined modularisation constructs. Additionally, interoperability between Event-B and other formalisms has been achieved in an ad hoc manner. Moreover, although a formal language, Event-B does not have a formal semantics. We address each of these limitations in this thesis using the theory of institutions.

The theory of institutions provides a category-theoretic way of representing a formalism. Formalisms that have been represented as institutions gain access to an array of generic specification-building operators that can be used to modularise specifications in a formalism-independent manner. In the theory of institutions, there are constructs (known as institution (co)morphisms) that provide us with the facility to create interoperability between formalisms in a mathematically sound way.

The main contribution of this thesis is the definition of an institution for Event-B, $\mathcal{EVT}$, which allows us to address its identified limitations. To this end, we formally define a translational semantics from Event-B to $\mathcal{EVT}$. We show how specification-building operators can provide a unified set of modularisation constructs for Event-B. In fact, the institutional framework that we have incorporated Event-B into is more accommodating to modularisation than the current state-of-the-art for Rodin. Furthermore, we present institution morphisms that facilitate in-

teroperability between the respective institutions for Event-B and UML. This approach is more generic than the current approach to interoperability for Event-B and in fact, allows access to any formalism or logic that has already been defined as an institution. Finally, by defining $\mathcal{EVT}$, we have outlined the steps required in order to include similar formalisms into the institutional framework. Hence, this thesis acts as a template for defining an institution for a specification language.

PUBLICATIONS

The publications that arose as a result of the research contained in this thesis are as follows:

Peer-Reviewed:

[40] Marie Farrell, Rosemary Monahan, and James F. Power. "Providing a Semantics and Modularisation Constructs for Event-B using Institutions". In: *23rd International Workshop on Algebraic Development Techniques, WADT*. 2016, pp. 18–19

[41] Marie Farrell, Rosemary Monahan, and James F. Power. "An Institution for Event-B". in: *Recent Trends in Algebraic Development Techniques. WADT 2016.* Vol. 10644. LNCS. 2017, pp. 104–119

[42] Marie Farrell, Rosemary Monahan, and James F. Power. "Specification Clones: An empirical study of the structure of Event-B specifications". In: *15th International Conference on Software Engineering and Formal Methods, SEFM*. vol. 10469. LNCS. 2017, pp. 152–167

[39] Marie Farrell, Rosemary Monahan, and James F. Power. "Modularising and Promoting Interoperability for Event-B Specifications using Institution Theory (Poster)". In: *28th European Summer School in Logic, Language and Information, ESSLLI*. 2016, p. 74

[36] Marie Farrell. "Using the Theory of Institutions to Integrate Software Models via Refinement". In: *PhD Symposium at the 12th Inter-*

national Conference on Integrated Formal Methods,PhD-iFM. 2016

[37] Marie Farrell, Rosemary Monahan, and James F. Power. "An Approach to Integrating Software Models via Refinement (Poster)". In: *ACM womENcourage*. 2014

Contributed:

[38] Marie Farrell, Rosemary Monahan, and James F. Power. "A Logical Framework for Integrating Software Models via Refinement". In: *British Colloquium for Theoretical Computer Science, BCTCS*. 2016

ACKNOWLEDGMENTS

First and foremost, I would like to thank my family and friends for their unfaltering support and encouragement throughout my Ph.D. studies. Thank you for always having faith in me and special thanks to Laura, Jen and Danielle for always being there for me.

My supervisors Dr. Rosemary Monahan and Dr. James F. Power deserve more gratitude than mere words can offer. Their patience, encouragement and difficult questions have shaped this thesis and I am grateful for every second of the time and the laughs that we have shared. I would like to extend my gratitude to Prof. Dominique Méry and Dr. Ionut Tutu for their helpful discussions around the institution for Event-B.

I would like to acknowledge the members the Principles of Programming research group (POP) at Maynooth University (Andrew Healy, Keith O' Dúlaigh and Zheng Cheng) for always being there to lend a hand over the past four years. In particular, I am eternally grateful to Dr. Hao Wu for his continued friendship, encouragement and advice.

Thank you to my proof readers (Keith N., Rob, Louis, Will, Laura and Trisha) for helping me along the way and special thanks to Kelly for the many journeys that we have shared to and from Maynooth University.

Finally, to Gary - thank you for your patience and encouragement throughout my Ph.D. studies. I am looking forward to our next adventure together, now that I am, in your words, *finally* finishing university.

DECLARATION

I confirm that this is my own work and the use of all material from other sources has been properly cited and fully acknowledged.

Marie Farrell
October 2017

Part 0

INTRODUCTION AND BACKGROUND MATERIAL

"Logic is justly considered the basis of all other sciences."

– Alfred Tarski

1

INTRODUCTION

In the 1930s, Alfed Tarski developed a theory of metamathematics - a mathematics for describing mathematics [117]. Later, Strachey and Scott applied similar techniques to the realm of programming languages by introducing a mathematical framework within which the semantics of all imperative programming languages could be defined, and thus denotational semantics was born [107]. Denotational semantics made it possible to determine whether any program, written in the language over which the semantics was defined, computed the function that it was intended to. The mathematical objects constructed using denotational semantics represent the meaning of a particular program but they do not offer a means for proving its correctness. Floyd-Hoare logic is a calculus for reasoning about the correctness of computer programs in terms of preconditions and postconditions (written as predicates) [63]. Edsger Dijkstra proposed a reformulation of Floyd-Hoare logic that assigns to each statement in the language a total function between predicates over the state space [31]. This became known as the predicate transformer semantics and it has provided a generic way to reason about the properties of software systems [30].

Software has become an integral part of our daily lives, from the cars that we drive to the medical devices that we use and as such, it is often catastrophic when it fails. The Ariane 5 launcher exploded due to a software error costing an estimated €350,000,000 in damages [22]. A software error caused the Therac-25 machines to give massive overdoses of radiation to six cancer patients resulting in three patients' deaths [74]. More recently, the Airbus A400M crash in 2015 that killed

all four crew members has been linked to a software fault. Each of these software failures could have been avoided if the appropriate techniques had been used to formally verify the systems at hand.

Formal methods provide a mechanism by which we can ensure the correctness of a given software system. Dijkstra's predicate transformer semantics was one of the first formal notations used for describing the *specification* of a program [30]. This allows us to *specify* a program P in terms of its preconditions and postconditions so that if the precondition is true then the execution of P must establish the postcondition. Along this vein, a myriad of languages have been developed allowing the user to formally and systematically verify that software systems preserve their pre and postconditions including VDM [68], Z [121], B [104], Event-B [3], Dafny [78], Spec# [12], JML [77], TLA+ [76] and CSP [64] to name but a few.

In this branch of formal software development, it is common to model software at different levels of abstraction, starting with a very high level abstract specification and finishing with a detailed concrete implementation. In formal software engineering we can map between these levels of abstraction in a verifiable way through a process known as *refinement* [8, 10, 86, 87, 96, 97]. The refinement calculus is a formal approach to program construction and consists of a notation and rules for deriving programs from their specifications [8, 10, 86, 87]. The theory of general refinement postulates that

> "The concrete entity C is a refinement of the abstract entity A when no user of A could observe if they were given C in place of A" [96, 97].

Formal refinement allows us to build up the system gradually and it offers a means by which proofs carried out in an abstract model of the system can be reused to prove properties about the concrete model. All of the languages outlined above (apart from Spec#) support the process

of formal refinement, and, in this thesis we focus specifically on Event-B.

Event-B is an industrial-strength, formal specification language used for proving the safety of a system's specification. The process of refinement is core to the Event-B methodology, and proof support is achieved via the Rodin platform, an Eclipse-based IDE for Event-B [3, 6]. Industrial applications of Event-B include air-traffic control systems, train interlocking systems and medical devices [70, 83, 112].

Tarski's work also paved the way for the development of institution theory which was introduced in the 1970s by Goguen and Burstall in a series of seminal papers [15, 50–52]. Institutions are described in terms of category theory, but draw on research in logical frameworks and abstract model theory [65, 84, 88]. A given institution is a collection of categories and functors, satisfying certain conditions, that acts as a description of a logic. More precisely, an institution defines the vocabulary, syntactic constructs, and meaning of validity within a particular logic [52]. Moreover, institutions provide a framework within which it is possible to formalise a given logic whilst providing access to an array of generic modularisation constructs and a framework for defining interoperability between them [16, 101]. Throughout this thesis, we investigate and apply these logic-based approaches in the setting of formal software engineering, specifically to the Event-B formal specification language. This provides a means for formalising its semantics, modularisation constructs and interoperability between Event-B and other formalisms in this framework.

1.1 MOTIVATION

The formal approach to software development relies on being able to describe properties of a program with a formally defined semantics,

typically logic-based, and being able to formally prove that these properties hold of a given model. The initial gulf between such formal methods and practical software engineering is increasingly bridged through Model Driven Engineering (MDE), with relatively lightweight formal methods, such as JML, Code Contracts and OCL becoming realistic elements of software development [35, 72, 77, 120]. Other modelling approaches, such as those based on formal specification, verification and model checking may also be used [66].

In this setting, there is a plethora of languages, based on different logics, specifying different aspects of a system, often at different levels of abstraction. Every individual language is inherently a formalisation of a particular (set of) logic(s), and so each can be referred to as a formalism, with the machinery provided by the language providing an accessible way to use the underlying logic. In Event-B the non-logic components of the language consist of the structuring constructs such as events, context extending and machine refining which will be discussed in detail in Section 2.1. Each of these formalisms is generally targeted at describing very specific kinds of properties about a system, such as safety or temporal properties for example. As systems increase in complexity it is more frequently necessary to verify more than one of these properties. This results in the same system being independently modelled in multiple, distinct formalisms, resulting in a time consuming and thus expensive process. An environment that facilitates interoperability between these formalisms in a provably correct way, would reduce the time and effort in verifying software systems and provide a solid, formal foundation to the theory of MDE. The theory of institutions provides an environment for interoperability between the logics that underlie these formalisms, and so, including the formalisms themselves into this framework would provide interoperability at a higher level of abstraction.

1.2 THESIS STATEMENT

Our core thesis is that the formalisms used for software verification, such as Event-B, can be represented as institutions, resulting in the creation of a heterogeneous environment where they can be used alongside each other using institution (co)morphisms [15, 52]. Specification-building operators supply a generic set of modularisation constructs that can be used by any formalism in this framework [100]. Another benefit is the definition of the semantics of each of these formalisms in a uniform way.

Therefore, we have developed an institution-theoretic formalisation of the Event-B specification language that provides a formal semantics, modularisation constructs and a framework for interoperability between Event-B and other formalisms. In general, interoperability between Event-B and other formalisms has been achieved in an ad hoc manner by building bespoke plugins for Rodin as will be discussed in Section 2.2.2. Our work provides a mathematical basis for the definition of such interoperability in a provable and correct way. Incorporating Event-B into the institutional framework involves specifying suitable categories and functors for the syntax and semantics of Event-B specifications, and verifying that they correctly meet the axiomatic requirements for institutional descriptions.

We have reconstructed Event-B case studies in the institutional context to exhibit the robustness of our approach. In particular, the successful integration of elements of specification from different formalisms (UML and Event-B) demonstrates the clear advantage of our approach over existing techniques.

1.3 SUMMARY OF CONTRIBUTIONS

The work described in this thesis has appeared in the following publications:

[40] Marie Farrell, Rosemary Monahan, and James F. Power. "Providing a Semantics and Modularisation Constructs for Event-B using Institutions". In: *23rd International Workshop on Algebraic Development Techniques, WADT*. 2016, pp. 18–19

[41] Marie Farrell, Rosemary Monahan, and James F. Power. "An Institution for Event-B". in: *Recent Trends in Algebraic Development Techniques. WADT 2016.* Vol. 10644. LNCS. 2017, pp. 104–119

[42] Marie Farrell, Rosemary Monahan, and James F. Power. "Specification Clones: An empirical study of the structure of Event-B specifications". In: *15th International Conference on Software Engineering and Formal Methods, SEFM*. vol. 10469. LNCS. 2017, pp. 152–167

The main theoretical contribution of this thesis is the definition of an institution for Event-B, $\mathcal{EVT}$ (Chapter 3) which yields three further advances [40, 41]. The first is the definition of a semantics for the entire Event-B formal specification language, including not only the basic mathematical language but also the structuring constructs (Chapter 4). The Heterogeneous Toolset, Hets [89], is the de facto standard tool providing support for formalisms represented as institutions and for some of the institution-theoretic avenues to interoperability. The Rodin tool supports Event-B and we have bridged the gap between these two environments by developing a translator, called EB2EVT, that uses the translational semantics defined in Chapter 4 to translate Event-B specifications to specifications in the $\mathcal{EVT}$ institution that we have prototyped in Hets.

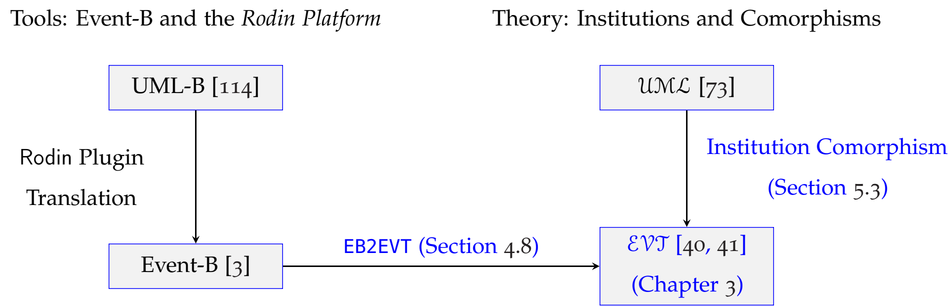


Figure 1.1: Interoperability can be achieved between UML and Event-B using their respective institutions and our EB2EVT translator tool.

Secondly, by incorporating the Event-B formal specification language into the institutional framework we provide scope for interoperability between Event-B and other formalisms that have been defined in this framework. In order to exploit this we have defined interoperability between UML and Event-B using their respective institutions and we illustrate the power of this by example in Chapter 5. The interoperability defined in Chapter 5 (institution comorphism) could have been implemented in HETS had the UML institution been included there. Our approach relied on the UML-B plugin, which translates UML state machines into Event-B, and EB2EVT to validate our comorphism [115]. We have illustrated the interoperability that can be achieved between UML and Event-B in Figure 1.1.

Finally, by working in the institutional framework we gain access to a set of generic specification-building operators that facilitate the modularisation of specifications in a formalism-independent manner. Additionally, we have contributed an empirical study of Event-B specifications that provided metrics for Event-B specifications and analysed the programming language concept of a code clone at the specification level. This study highlights the need for these modularisation constructs and shown, by example, how they can be applied to Event-B specifications (Chapter 6) [42].

Further to this we note that the theory of institutions had, until now, been predominantly used for representing logics, such as first-order logic, modal logic etc. [100]. In this thesis, we illustrate that the institutional approach can be applied to state-based formal specification languages, like Event-B, that are used in an industrial setting and furnish improvements to the way in which specifications can be written. We anticipate that this thesis can form a handbook for those wishing to incorporate other languages used for formal verification into this framework.

These contributions constitute the body of this thesis with Chapter 2 supplying the relevant background information and Chapter 7 summarising these contributions.

2

BACKGROUND MATERIAL

The sub-title of this thesis is "Defining a semantics, modularisation constructs and interoperability for a specification language" and in this chapter we will introduce each of these concepts and discuss how they are handled in both Event-B and the theory of institutions.

2.1 EVENT-B

Event-B evolved from the B-method (Classical-B) and it is an industrial-strength, state-based formal specification language for system-level modelling and verification for predominantly safety-critical systems [4]. It uses a notation based on set theory to model the specification of a system through a series of events that may be triggered nondeterministically. The objective of an Event-B specification is to provide a basis for proving properties of a given specification. An Event-B specification typically consists of *Machines* and *Contexts*. Machines (Definition 21) model the dynamic parts of a system's specification and typically contain event declarations (Definition 22) which model the state update. Contexts model the static properties of a system's specification (Definition 23).

Definition 21 (Event-B Machine). An Event-B machine specification is an 8-tuple $\langle id, RM, C, V, I, T, N, E\rangle$ where *id* is the machine identifier. The optional `refines` clause indicated by *RM* contains the machine identifier of the machine that the current machine refines. The optional set C contains the identifiers of the contexts that this machine `sees`. V is a set

of variable declarations, I is a set of invariant declarations (predicates), T is a set of theorems to be proven, N is a variant expression that is used for proving termination properties and E is a set of event declarations.

Definition 22 (Event-B Event). An Event-B event declaration is a 7-tuple $\langle id, S, RE, P, G, W, BA \rangle$ where id is the event identifier and S is the event status (either `ordinary`, `anticipated` or `convergent`). The optional set RE contains the event identifiers of the events that the current event `refines`. P is a set of event parameters (local variables), G is the event guard (predicate), W is an optional set of witnesses (predicates) that refine the parameters of the abstract event(s) listed in RE, and BA is the set of action statements that are represented internally as before-after predicates[1].

Definition 23 (Event-B Context). An Event-B context specification is a 5-tuple $\langle id, X, S, C, A \rangle$ where id is the context identifier. The optional set X contains the context identifiers for the contexts that the current one `extends`. S is a set of carrier set declarations, C is a set of constant declarations and A is a set of axioms.

One of the main differences between the structure of specifications written using the B method and those in Event-B is that Event-B distinguishes between machines and contexts whereas B does not. Moreover, operations in B can call other operations and in Event-B events are executed in a nondeterministic manner. In particular, refinement in Event-B is more general, enabling the addition of new events and a single event may be refined by many events [4].

Event-B is based on Back's Action Systems and supports two kinds of refinement: data refinement and superposition refinement [9]. We

1 The only exception to Definition 22 is the `Initialisation` event which cannot have parameters, guards or witnesses [6]. All Event-B machines must have an `Initialisation` event.

illustrate the structure of Event-B specifications and refinement using a simple example.

2.1.1 EXAMPLE: TRAFFIC-LIGHTS SYSTEM

Lines 1–21 of Figure 2.1 contain an Event-B machine, `mac1`, for a traffic-lights system with one light signalling cars and one signalling pedestrians [6]. The goal of the specification is to ensure that it is never the case that both cars and pedestrians receive the "go" signal at the same time (represented by boolean flags on line 2). This machine specification contains variable declarations (line 2), invariants (lines 3–6) and event specifications (lines 7–21).

The machine `mac1` specifies five different events (including the `Initialisation` event defined on lines 8–10). As outlined in Definition 22, each of these events has a guard, specifying when it can be activated, and an action, specifying what happens when the event is activated. For example, the `set_peds_go` event, as specified on lines 11–13, has one guard expressed as a boolean expression (line 12), and one action, expressed as an assignment statement (line 13). Events are essentially predicate transformers that describe preconditions (guards) and post-conditions (actions) with the action statement represented internally as a before-after predicate that relates the values of the variables before the event to the values of the variables after the event. For example, the assignment `x := x + 1` is interpreted as the before-after predicate `x′ = x + 1`, where the primed variable, `x′` denotes the after value of the variable `x`. Moreover, each event has a status, which can be either `ordinary`, `convergent`, or `anticipated`.

As outlined in Definition 21, machines can have a *variant* (natural number) expression that is used to facilitate proving termination properties. Events with a status of `anticipated` must not increase the variant

```
1  MACHINE mac1
2  VARIABLES  cars_go, peds_go
3  INVARIANTS
4     inv1: cars_go ∈ BOOL
5     inv2: peds_go ∈ BOOL
6     inv3: ¬ (peds_go = true
   ∧ cars_go = true)
7  EVENTS
8    Initialisation ordinary
9      then  act1: cars_go := false
10           act2: peds_go := false
11   Event set_peds_go ≙ ordinary
12     when  grd1: cars_go = false
13     then  act1: peds_go := true
14   Event set_peds_stop ≙ ordinary
15     then act1: peds_go := false
16   Event set_cars_go ≙ ordinary
17     when  grd1: peds_go = false
18     then  act1: cars_go := true
19   Event set_cars_stop ≙ ordinary
20     then act1: cars_go := false
21 END

22 CONTEXT ctx1
23 SETS  COLOURS
24 CONSTANTS  red, green, orange
25 AXIOMS
26    axm1: partition(COLOURS,
   {red}, {green},{orange})
27 END
```

```
28 MACHINE mac2  refines mac1   SEES ctx1
29 VARIABLES  cars_colour, peds_colour,
30            buttonpushed
31 INVARIANTS
32    inv1: peds_colour ∈ {red,green}
33    inv2: (peds_go = true)
   ⇔ (peds_colour = green)
34    inv3: cars_colour ∈ {red,green}
35    inv4: (cars_go = true)
   ⇔ (cars_colour = green)
36    inv5: buttonpushed ∈ BOOL
37 EVENTS
38   Initialisation ordinary
39     then  act1: cars_colour := red
40           act2: peds_colour := red
41   Event set_peds_green ≙ ordinary
42   refines  set_peds_go
43     when  grd1: cars_colour = red
44           grd2: buttonpushed = true
45     then  act1: peds_colour := green
46           act2: buttonpushed := false
47   Event set_peds_red ≙ ordinary
48   refines  set_peds_stop
49     then  act1: peds_colour := red
50   Event set_cars_green ≙ ordinary
51   refines  set_cars_go
52     when  grd1: peds_colour = red
53     then  act1: cars_colour := green
54   Event set_cars_red ≙ ordinary
55   refines  set_cars_stop
56     then  act1: cars_colour := red
57   Event press_button ≙ ordinary
58     then  act1: buttonpushed := true
59 END
```

Figure 2.1: Event-B specification of a traffic system. Lines 1–21 contain an Event-B specification for an abstract machine that uses boolean flags to describe the behaviour of the traffic system. The Event-B context on lines 22–27 provides a specification for the colours of a set of traffic–lights. Lines 28–59 contain a refined Event-B machine specification for a traffic system.

expression and those with a status of `convergent` must strictly decrease the variant expression. Our example has no variant so all events have the status `ordinary`.

Lines 22–27 of Figure 2.1 contain a context specification for the data-type COLOURS. The axiom on line 26 explicitly restricts the carrier set of COLOURS to only contain the constants *red*, *green* and *orange*.

Event-B supports the process of refinement and lines 28–59 of Figure 2.1 contains an Event-B machine specification for the machine `mac2` that

refines the machine `mac1` (lines 1–21 of Figure 2.1). The machine `mac2` refines `mac1` by first introducing the new context (as indicated by the `sees` clause on line 28 and then by replacing the truth values used in the abstract machine with new values from the carrier set COLOURS. This new data type is used by the new variables *cars_colour* and *peds_colour* on line 29 of Figure 2.1.

During refinement, the user typically supplies a *gluing invariant* relating properties of the abstract machine to their counterparts in the concrete machine [6]. The gluing invariants in Figure 2.1 (lines 33 and 35) define a one-to-one mapping between the concrete variables introduced in `mac2` and the abstract variables of `mac1`. The concrete variables (*peds_colour* and *cars_colour*) can be assigned a value of either *red* or *green*, thus the gluing invariants map the values of the concrete variable to the values of the abstract variables (*true* to *green* and *false* to *red*). This form of refinement is referred to as *data refinement* [3].

Superposition refinement in Event-B permits the addition of new variables and events [3] such as `button_pushed` (line 30) and `press_button` (lines 57–58) in Figure 2.1. The existing events from `mac1` are renamed to reflect refinement; for example, the event `set_peds_green` is declared to refine `set_peds_go` (lines 41–42). This event has also been altered via the addition of a guard (line 44) and an action (line 46) that incorporate the functionality of a button-controlled pedestrian light. This example highlights the features of the Event-B language and we will refer to it in the chapters that follow.

Event-B is used for proving the safety of a given specification and in Section 2.1.2 we describe the tool support for proving these properties.

2.1.2 TOOL SUPPORT AND PROOF

Proof support for Event-B specifications is provided by the Rodin Platform, an Eclipse-based IDE for Event-B, by generating a series of proof obligations for a given Event-B specification [3]. The main safety proof obligations consist of invariant preservation as invariants are required to hold before and after every event [6]. Refinement is core to the Event-B methodology, as such, a number of proof obligations are generated that are specific to refinement. This allows the developer to establish that the refinement steps that they have taken are correct [3, 5]. Other proof obligations include well-definedness of expressions, termination (using the variant expression) and feasibility of actions.

These proof obligations can be proven automatically or interactively via the Rodin proving perspective with the Atelier-B theorem prover typically providing proof support [81]. Other theorem provers (such as Isabelle) have also been integrated as Rodin plugins [103]. Historically, the semantics of Event-B specifications is understood in terms of the corresponding proof obligations that are generated by Rodin [59]. This offers the benefit of allowing Event-B to be used for specifying systems in various modelling domains (cyber-physical systems, algorithms etc.), however, it does not offer semantics for Event-B models. In Chapter 4, we provide a formally defined translational semantics for Event-B, using the theory of institutions, that does not inhibit this flexibility. Apart from its lack of a formally defined semantics, we have identified and summarised the limitations of the Event-B formal specification language in Section 2.2.

2.2 LIMITATIONS OF EVENT-B

Although a mature specification language, based on our experience with Event-B and observing case studies, we have identified two primary limitations of Event-B.

1. Event-B lacks well-developed *modularisation* constructs as indicated by the plethora of plugins in Table 2.1. Notice how, in Figure 2.1 the same specification has to be provided twice in the machine `mac1`. Here, the events `set_peds_go`, and `set_peds_stop` are equivalent, modulo renaming of variables, to `set_cars_go` and `set_cars_stop`.

2. When developing software using Event-B, it is at least necessary to transform the final concrete specification into a different language to get an executable implementation [112]. Current approaches to *interoperability* in Event-B consist of a range of Rodin plugins to translate to/from Event-B, but these often lack a solid logical foundation.

In Sections 2.2.1 and 2.2.2, we examine the current approaches to addressing these pitfalls in specification languages, and specifically, in the Event-B specification language.

2.2.1 MODULARISATION AS DECOMPOSITION IN EVENT-B

As systems increase in both size and complexity, the more evident it becomes that a "divide and conquer" approach to software engineering is necessary, particularly in the field of safety critical systems. The standard approach to solving this problem is by adopting a modular approach to software development. In modular software engineering, a system is decomposed into multiple independent *modules* that can

be recomposed at a later stage to form the entire system [48]. There are a multitude of benefits to this approach including an ease of code maintenance, code reuse and enabling multiple engineers to work on different components of the same system in parallel. In general, formal specification languages utilise refinement as their main modularisation technique with support for other modularisation features typically added after the language has been defined [34, 69]. This generally results in reengineering the specification language, as was the approach taken with the Z specification language, resulting in Object-Z, in order to provide scope for modular developments [113, 121]. Similarly, the VVSL language was developed as an approach to modularisation for VDM [85].

During the evolution of the Event-B formalism from Classical-B, certain facilities for the reuse of machine specifications disappeared. These include the modularisation properties supplied by the keywords `INCLUDES` and `USES` which facilitated the use of an existing machine in other developments [108]. Since then there have been numerous attempts at regaining such modularisation techniques [17, 53, 54, 60, 95, 109]. Modularisation in Event-B does not require reengineering the language in the same way as Object-Z or VVSL, but rather, modifications are made by building plugins for Rodin and we examine these techniques here.

Modularisation, in terms of decomposition, in Event-B was first described by Abrial as the act of cutting a large system of events into smaller pieces that can each be refined independently of the others [5]. This offers the advantage of separation of concerns even though there still needs to be a link between the parts. The central idea is that the specifications constructed via decomposition and further refinement may be easily recombined resulting in a specification that could have been obtained without using any decomposition techniques in the first place. Three primary types of Event-B decomposition have been identi-

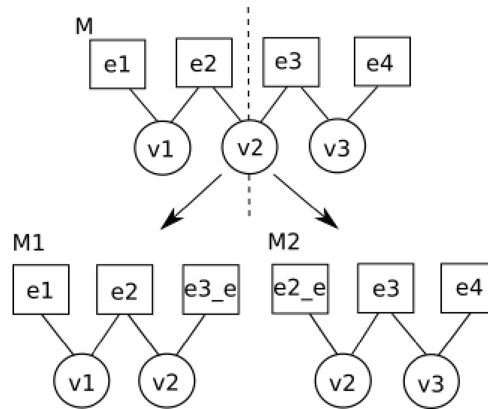


Figure 2.2: The shared variable decomposition of machine M into submachines M1 and M2.

fied [60]. These are *shared variable* [2], *shared event* [17] and *modularisation* [67]. Another related approach is *generic instantiation* [5]. These four approaches are further summarised below.

Shared Variable Decomposition

Using the *shared variable* approach, an Event-B machine is decomposed into sub-machines based on events sharing the same variables as can be seen in Figure 2.2 [5]. Since it is possible for multiple events to refer to the same variable (e2 and e3 both refer to v2 in Figure 2.2) external events are added (e3_e and e2_e in Figure 2.2) to abstractly describe the behaviour of the shared variable in the other submachine(s). Neither these shared variables nor external events may be refined in their respective machine(s). Although there is no intention to recompose these submodels, it is theoretically possible to do so [5]. This approach enables the user to independently refine submodels and facilitates several developers working on different submodels of the same large system at the same time [2].

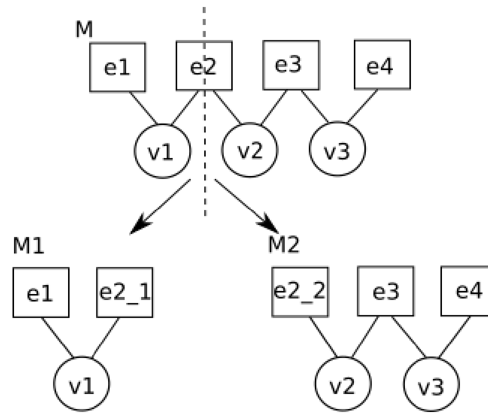


Figure 2.3: The shared event decomposition of machine M into submachines M1 and M2.

Shared Event Decomposition

Shared event decomposition partitions an Event-B machine based on variables which participate in the same events, this is illustrated in Figure 2.3 [5]. Here, the variable v1 participates in events e1 and e2. The variables v2 and v3 participate in events e2, e3 and e4. The partitioning of this machine places the events that refer to v1 in one machine and those referring to the other variables in another. Both sets of variables {v1} and {v2, v3} participate in event e2, thus, e2 is decomposed into two events e2_1 and e2_2. Respectively, these new events each correspond to a restricted version of the original event e2, each confined to the guards and actions that refer to each of the variable sets (i.e. e2_1 contains only the guards and actions of e2 that referenced v1). In this way, the machine M is decomposed into submachines M1 and M2 such that $M = M1 \| M2$ where $\|$ denotes CSP-style parallel composition [64]. M1 and M2 do not share any common variables and they interact over shared events which are events with the same name [17]. When combining these machines the guards are conjoined and actions of each shared event are composed in parallel. Unlike the shared variable approach, data refinement is allowed on all variables as there are no shared variables. It is also possible to inde-

pendently refine all shared events although the user must be careful to avoid unintentional naming conflicts.

Modularisation

The *modularisation* approach to decomposition is illustrated in Figure 2.4 and is considered a special case of the shared variable approach [67]. Here, *module interfaces* (denoted by `I`) define operations as pairs of preconditions and postconditions and they can also contain interface variables (`IV`). Module interfaces may "see" contexts and contain module invariants but they do not contain any events. The *module implementation* (`IM`) contains groups of events (these are denoted by `g1`, `g2` and `g3`). Each *event group* corresponds to the implementation of one operation (`o1` or `o2`) in the module interface and vice versa. Events can call operations and then use the returned results. In this case, a set of rewrite rules are used to translate the operation call back to standard Event-B. This translation is not observed by the user but is used to generate the corresponding proof obligations. Although based upon the shared variable approach which was initially aimed at distributed systems, the modularisation approach splits the functionality of a sequential system into several modules that can be developed independently and in parallel [67].

Generic Instantiation

Generic instantiation facilitates the parametrisation of machines in order to reuse refinements [5]. A machine refinement chain is generic with respect to all of the carrier sets and axioms that have been collected in the contexts used throughout. It is possible that, in some (new) Event-B model, a developer can reuse another (previous) Event-B model by instantiating the sets and constants of the old model. In order to reuse the proofs of the old model, its sets, constants and axioms are proven as

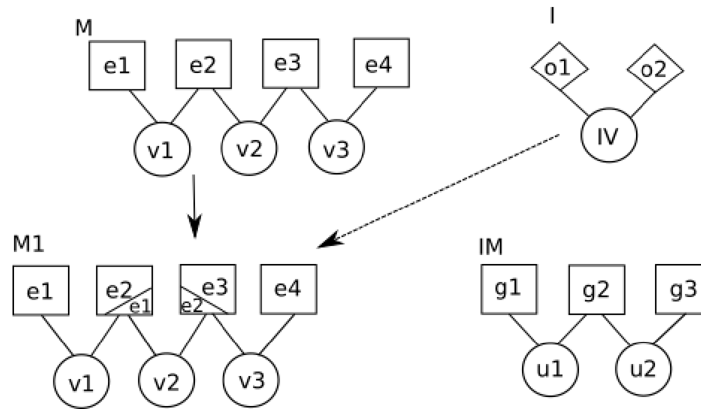


Figure 2.4: The modularisation approach to decomposition using module interfaces and event groups.

theorems after instantiation in the new model. This process facilitates the reuse of existing Event-B models thus providing a more modular approach to specification in Event-B.

An array of Rodin plugins have been constructed that offer modularisation features and most of these are based on the approaches outlined above. We have summarised the relevant plugins in Table 2.1 and in Appendix A, we show how each of them can be captured in the theory of institutions.

It is clear, from observations of industrial projects and the sheer volume of Rodin plugins developed for modularisation in Event-B, that there is an underlying demand for modularisation [60, 67]. We investigate this further in Chapter 6 by studying the frequency of specification clones throughout a corpus of Event-B specifications (an approach that we adopted from the code clone literature [98]).

Furthermore, although all of these plugins offer some form of modularisation for Event-B specifications, it is not clear how specifications

2 http://wiki.event-b.org/index.php/XEvent-B

3 http://wiki.event-b.org/index.php/Refactoring_Framework

Plugin Name	**Description of functionality**
Feature Composition	*Features* are comprised of machines and their (seen) contexts, they are combined by *fusing* variables and events. There are three distinct functionalities of this plugin: (1) composition of machines, (2) making machines disjoint before composition and (3) partial composition of machines [53–55, 95].
Generic Instantiation	Enables the reuse of generic Event-B models in other Event-B models and provides a mechanism for extending the instantiation to a chain of refinements. Here, the "pattern" machine is instantiated in order to refine the "problem" machine [108]. Instantiation is achieved by parametrising contexts and using the concept of a "sharing context" to allow a context be seen by several machines. Special proof obligations are generated to ensure that the instantiation of the pattern is valid.
Model Decomposition	Based on the shared variable/shared event approach discussed earlier, the user selects the machine to be decomposed and defines the subcomponents (machines and contexts) to be generated [109]. Then they select the style of decomposition to use (shared variable (A-style) or shared event (B-style)) and can opt to decompose the contexts in a similar fashion. These generated subcomponents can then be further refined.
Pattern	Provides design patterns for Event-B that facilitate the reuse of an existing Event-B model (*pattern*) in the current Event-B model (*problem*) [47]. If at some point during development the developer realises that the current model closely matches one that they have already completed as part of another project, then they match this pattern and can incorporate it into the current problem by carrying out some renamings.
Shared Event (Parallel) Composition	The composition is based on that proposed by Butler [17]. It uses CSP-style parallel composition $\parallel$ to compose machines through events. This is the precursor to event refinement structures (ERS) which are sometimes referred to as atomicity decomposition [43].
Modularisation	Inspired by the modularisation approach that was outlined earlier, *modules* split up an Event-B component (machine/context) and are paired with an interface defining the conditions for incorporating one module into another [67].
XEvent-B	Provides a way of combining machines called *machine inclusion*[2].
Theory	Facilitates the addition of new data types, operators and axioms ("theories") for use in multiple independent Event-B models [18].
Renaming Refactory	Supports the renaming of variables, parameters, carrier sets, constants, events and other labelled elements (invariants, axioms, guards, etc) [3].

Table 2.1: Rodin plugins that employ modularisation features.

developed utilising more than one of these plugins could be combined in practice, or, if this is even possible.

2.2.2 INTEROPERABILITY AND HETEROGENEITY

Fundamentally, the application of formal methods to a system can involve the use of multiple formalisms, as outlined in Section 1.1, either because parts of the system are more agreeable to one formalism over another or because of the sheer complexity of the system. By *interoperability*, we mean the integration of multiple formalisms to ensure the correctness of the overall system. One approach to interoperability is to provide a translation from each of the individual languages to one underlying formalism. This is the approach taken by the group of verification languages (Spec# [12], Dafny [78], Chalice [79], etc.) that all translate to the Boogie intermediate verification language which uses the Z3 solver for verification [13]. Similarly, the Why3 platform promotes the interoperability of a range of theorem provers via the WhyML intermediate verification language [45].

The usual way of providing interoperability for Event-B is by employing a Rodin plugin to generate the desired code from the Event-B specification [19–21, 33, 93, 111, 119, 122]. A series of plugins offering this functionality have been developed for Rodin and we provide an outline of those relevant to this discussion in Table 2.2. These plugins generally offer a bespoke translation from Event-B to another formalism but the translation itself can be somewhat ad hoc in nature.

For example, the EventB2Java and EB2ALL plugins both generate Java programs but the code that they generate is very different. EventB2Java generates JML specifications alongside a multi-threaded Java implementation, whereas, EB2ALL does not include JML specifications and the

4 http://wiki.event-b.org/index.php/B2Latex

5 http://wiki.event-b.org/index.php/Transformation_patterns

Plugin	Description of functionality
UML-B	Generates Event-B models from UML state machines and class diagrams [115].
B2Latex	Generates Latex from Event-B specifications[4].
EventB2Java	Generates Java implementations with JML specifications from Event-B models [20].
EventB2JML	Generates JML-specified abstract Java classes [21].
EventB2Dafny	Translates Event-B proof obligations to Dafny [19].
EventB2SQL	Generates Java implementations of Event-B machines that store the state of a machine in a database [119].
EB2ALL	Generates C, C++, Java and C# from Event-B [111].
Tasking Event-B	Generates multi-tasking Java, Ada and OpenMP C from Event-B [33].
B2C	Translates Event-B specifications to C code [122].
EHDL	Supports VHDL code generation from Event-B models [93].
MBT Plugin	Generates test sequences that cover the events of an Event-B model [32].
Transformation Patterns	Supports writing and running transformation scripts over Event-B models in EOL[5].

Table 2.2: Rodin plugins that support interoperability for Event-B.

Java programs that it generates are sequential [20, 82]. The EventB2SQL plugin also generates Java implementations but, this plugin again, follows a different approach for its translation as it stores the state of an Event-B machine in an SQL database [119]. Each of these independent translations can be advantageous from the perspective of the developer, but one might ask: are the corresponding Java implementations equivalent, and, if so, why are there multiple plugins performing the same task?

In theory, proving equivalence would require a fully formal semantics for the portion of the Java language being targeted. Even with such a semantics, there are two levels at which we can consider such an equiv-

alence: (1) between the Event-B specification and the Java program and, (2) between the Java programs generated by each of the plugins. In the first case, there is no way of measuring the equivalence between the Event-B specifications and the corresponding Java code, since the nondeterministic nature of Event-B events must be resolved using scheduling in Java. Thus, the closest that we can come to measuring such an equivalence is by maximally obtaining a simulation relation where any step of the program can be done by the specification but not vice versa, in fact, this is a refinement relation.

In the second case, we recognise that the developers of each of these plugins envisaged a different end product. The EventB2Java plugin generates a Java implementation of an Event-B model that can undergo further verification using JML, and the non-deterministic behaviour of event triggering is implemented using multiple threads. In constrast, the Java translation of EB2ALL forms part of a larger toolkit that can also generate C, C++ and C#, with no further verification anticipated. In this case, event triggering is implemented via a series of `if-else` statements. Finally, the EventB2SQL plugin is designed so that Event-B machines can be used to model and generate Java implementations of database applications. It is thus clear that each of these translations is bespoke, and so the translations are not equivalent, although very useful in different scenarios.

Another approach to interoperability is the writing of heterogeneous specifications where specifications written in different formalisms can be reasoned about alongside one another [57]. Although beneficial, this is generally difficult to achieve in practice, and often involves the development of an entirely new toolset, such as Circus which provides an environemnt for specifiying and verifying combinations of Z and CSP [44]. Some Rodin plugins, such as UML-B, support a heterogeneous approach to specification in Event-B by specifying UML state machines

alongside Event-B models [115]. Another formalism that supports heterogeneity in this setting, but has not been represented as a plugin, is the combined formalism Event-B||CSP [105]. We will discuss the UML-B heterogeneous approach to interoperability in more detail in Chapter 5 when we will use the theory of institutions to provide a mathematical foundation for the interoperability provided by the UML-B plugin.

We believe that the theory of institutions can address the identified pitfalls of modularisation and interoperability in Event-B, and we discuss this in Section 2.4. However, we first provide some of the mathematical prerequisites of this theory in the Section 2.3.

2.3 THE THEORY OF INSTITUTIONS

The theory of institutions, originally developed by Goguen and Burstall in a series of papers originating from their work on algebraic specification, provides a general framework for defining a logical system [15, 50, 52]. Institutions are generally used for representing logics in terms of their vocabulary (signatures), syntax (sentences) and semantics (models and satisfaction condition). The basic maxim of institutions, inspired by Tarski's theory of truth [116], is that

> *"Truth is invariant under change of notation"* [52].

The use of institutions facilitates the development of specification language concepts, such as structuring of specifications, parametrisation, and refinement. This is achieved completely independent of the underlying logical system resulting in an environment for heterogeneous specification and combination of logics [15, 101]. This was originally illustrated in the definition of the Clear algebraic specification language [16], and, more recently in the Common Algebraic Specification Language (CASL) [90] and the Web Ontology Language (OWL) [7].

Goguen and Burstall also developed the notions of "charters" and "parchments" as a way to generate institutions using the fact that the syntax of a logical system forms an initial algebra [51]. However, this approach does not receive much attention in the literature.

As the theory of institutions finds its foundations in category theory, we provide some basic category theoretic definitions to frame our description of institution theory in Section 2.3.1. We then present the formal definition of an institution (Definition 28) and describe the institution for first-order predicate logic with equality, $\mathcal{FOPEQ}$ (Section 2.3.3). Furthermore, we discuss refinement in the institutional setting and its correspondence with refinement in Event-B in Section 2.3.4.

2.3.1 CATEGORY-THEORETIC PREREQUISITES

Category theory, similar to set theory, provides a fundamental framework for reasoning which has been applied across a range of disciplines in computer science [49, 99, 118]. In set theory we examine objects, their membership in a particular set and ways of reasoning about these sets. Category theory abstracts away from the objects and allows us to reason about the relationships between them. Category theory and set theory are not contradictory but rather they are alternatives, each with a different emphasis [100]. From the perspective of institutions, category theory provides a foundational framework for algebraic semantics and specification languages.

A *category* (Definition 24) consists of a collection of objects, morphisms between them and a morphism composition operation, '$\circ$', that must preserve certain properties.

Definition 24 (Category). A category $\mathbf{K}$ consists of

A collection $|\mathbf{K}|$ of $\mathbf{K}$-objects.

For each A, $B \in |\mathbf{K}|$, a collection $\mathbf{K}(A, B)$ of $\mathbf{K}$-morphisms from A to B.

For each $A, B, C \in |\mathbf{K}|$, a composition operation

$$_ \circ _ : \mathbf{K}(A, B) \times \mathbf{K}(B, C) \to \mathbf{K}(A, C)$$

such that the following properties hold.

1. for all $A, B, A', B' \in |\mathbf{K}|$, if $\langle A, B\rangle \neq \langle A', B'\rangle$ then

$$\mathbf{K}(A, B) \cap \mathbf{K}(A', B') = \varnothing$$

2. (existence of identities) for each $A \in |\mathbf{K}|$ there is a morphism $id_A \in \mathbf{K}(A, A)$ such that $id_A \circ g = g$ for all morphisms $g \in \mathbf{K}(A, B)$ and $f \circ id_A = f$ for all morphisms $f \in \mathbf{K}(B, A)$.
3. (associativity of composition) for any $f \in \mathbf{K}(A, B), g \in \mathbf{K}(B, C)$ and $h \in \mathbf{K}(C, D), f \circ (g \circ h) = (f \circ g) \circ h$.

Example: **Set** is the category of sets where objects are sets and morphisms are set-theoretic functions. In this setting the composition of morphisms is function composition. This category preserves the required properties 1,2 and 3 presented in Definition 24. Note that when describing categories, morphisms are often referred to as *arrows*.

Functors (Definition 25) describe a mappings between categories with objects mapped to objects and morphisms mapped to morphisms.

Definition 25 (Functor). A *functor* $\mathbf{F} : \mathbf{K1} \to \mathbf{K2}$ from a category $\mathbf{K1}$ to a category $\mathbf{K2}$ consists of:

A function $\mathbf{F}_{Obj} : |\mathbf{K1}| \to |\mathbf{K2}|$.

For each $A, B \in |\mathbf{K1}|$, a function $\mathbf{F}_{A,B} : \mathbf{K1}(A, B) \to \mathbf{K2}(\mathbf{F}_{Obj}(A), \mathbf{F}_{Obj}(B))$.

such that:

1. $\mathbf{F}$ preserves identities: $\mathbf{F}_{A,A}(id_A) = id_{\mathbf{F}_{Obj}}(A)$ for all objects $A \in |\mathbf{K}|$.

2. **F** preserves composition: for all morphisms $f : A \to B$ and $g : B \to C$ in **K1**, $\mathbf{F}_{A,C}(f \circ g) = \mathbf{F}_{A,B}(f) \circ \mathbf{F}_{B,C}(g)$.

Example: The power set functor $\mathbf{P} : \mathbf{Set} \to \mathbf{Set}$ maps each set to its power set and each function $f : X \to Y$ to the map which sends $Z \subseteq X$ to $f(Z) \subseteq Y$ [100].

In general, the term "forgetful functor" is used to refer to any functor that, intuitively, forgets the structure of objects in a category, mapping any structured object to its underlying unstructured set of elements. An example of a forgetful functor is the functor that maps any topological space to the set of its points.

Definition 26 (Functor Composition)**.** If $\mathbf{F} : \mathbf{K1} \to \mathbf{K2}$ and $\mathbf{G} : \mathbf{K2} \to \mathbf{K3}$ are functors, then their composition, denoted by $\mathbf{F};\ \mathbf{G} : \mathbf{K1} \to \mathbf{K3}$, is a functor such that: $(\mathbf{F};\ \mathbf{G})_{Obj} = \mathbf{F}_{Obj};\ \mathbf{G}_{Obj}$ and $(\mathbf{F};\ \mathbf{G})_{A,B} = \mathbf{F}_{A,B};\ \mathbf{G}_{\mathbf{F}(A),\mathbf{F}(B)}$ for all $A, B \in \mathbf{K1}$ [100].

Categories and functors form the basic building blocks of an institution as will be illustrated in Definition 28. However, for the purposes of interoperability between institutions, *natural transformations* are required. A natural transformation defines a mapping between two functors and these will be used to map between the syntax and semantics parts of an institution as will be described in Section 2.4.1. Definition 27 formalises the category-theoretic concept of a natural transformation.

Definition 27 (**Natural Transformation**)**.** Given categories **K1** and **K1**, and functors $\mathbf{F} : \mathbf{K1} \to \mathbf{K2}$ and $\mathbf{G} : \mathbf{K1} \to \mathbf{K2}$. A natural transformation from **F** to **G**, is a family $\langle \tau_A : \mathbf{F}(A) \to \mathbf{G}(A) \rangle_{A \in |\mathbf{K1}|}$ of **K2**-morphisms such that for any $A, B \in |\mathbf{K1}|$ and **K1**-morphism $f : A \to B$ the following diagram commutes, that is $\tau_A;\ \mathbf{G}(f) = \mathbf{F}(f);\ \tau_B$:

K1 :

$$\begin{array}{c} A \\ \downarrow f \\ B \end{array}$$

K2 :

$$\begin{array}{ccc} \mathbf{F}(A) & \xrightarrow{\tau_A} & \mathbf{G}(A) \\ \downarrow \mathbf{F}(f) & & \downarrow \mathbf{G}(f) \\ \mathbf{F}(B) & \xrightarrow{\tau_B} & \mathbf{G}(B) \end{array}$$

We have outlined the necessary category-theoretic prerequisites for the theory of institutions and in the next sections we define what is meant by an institution and how they can be used to address the limitations of Event-B.

2.3.2 INSTITUTIONS

Based on Definitions 24 and 25, we define an institution as follows [52]:

Definition 28 (Institution). An **institution** $\mathcal{INS}$ for some given logic consists of

A category **Sign** whose objects are called *signatures* and whose arrows are called *signature morphisms*.

A functor $\mathbf{Sen} : \mathbf{Sign} \to \mathbf{Set}$ yielding a set $\mathbf{Sen}(\Sigma)$ of Σ-*sentences* for each signature $\Sigma \in |\mathbf{Sign}|$ and a function $\mathbf{Sen}(\sigma) : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$ for each signature morphism $\sigma : \Sigma \to \Sigma'$.

A functor $\mathbf{Mod} : \mathbf{Sign}^{op} \to \mathbf{Cat}$ yielding a category $\mathbf{Mod}(\Sigma)$ of Σ-*models* for each signature $\Sigma \in |\mathbf{Sign}|$ and a functor $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$ for each signature morphism $\sigma : \Sigma \to \Sigma'$.

For every signature Σ, a *satisfaction relation* $\models_{\mathcal{INS},\Sigma}$ determining satisfaction of Σ-sentences by Σ-models.

An institution must uphold the **satisfaction condition**: for any signature morphism $\sigma : \Sigma \rightarrow \Sigma'$, translations $\mathbf{Mod}(\sigma)$ of models, $\mathbf{Sen}(\sigma)$ of sentences, $\phi \in \mathbf{Sen}(\Sigma)$ and $M' \in |\mathbf{Mod}(\Sigma')|$ then

$$M' \models_{\mathcal{INS},\Sigma'} \mathbf{Sen}(\sigma)(\phi) \quad \Leftrightarrow \quad \mathbf{Mod}(\sigma)(M') \models_{\mathcal{INS},\Sigma} \phi \tag{2.1}$$

where $\mathbf{Sen}(\sigma)(\phi)$ (resp. $\mathbf{Mod}(\sigma)(M')$) indicates the application of the signature morphism σ to the sentence ϕ (resp. model M').

Note that we often denote $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma') \rightarrow \mathbf{Mod}(\Sigma)$ by $_|_\sigma$: $\mathbf{Mod}(\Sigma') \rightarrow \mathbf{Mod}(\Sigma)$ to indicate the model reduct along a signature morphism $\sigma : \Sigma \rightarrow \Sigma'$. Model reducts are central to the definition of an institution as they allow us to consider a model over one signature as a model over another via a signature morphism. They are also core to institution-theoretic refinement as discussed in Section 2.3.4. Fundamentally, specifications consist of a number of sentences, therefore, we provide the following definition of a *presentation* to formally define how these sentences are combined.

Definition 29 (Presentation)**.** For any signature Σ, a Σ-presentation (sometimes called a *flat specification*) is a pair $\langle \Sigma, \Phi \rangle$ where $\Phi \subseteq \mathbf{Sen}(\Sigma)$. $M \in |\mathbf{Mod}(\Sigma)|$ is a model of a Σ-presentation $\langle \Sigma, \Phi \rangle$ if $M \models \Phi$ [100].

In this section, we have defined the concept of an institution and in Section 2.3.3, we present the institution for first-order predicate logic with equality.

2.3.3 EXAMPLE: $\mathcal{FOPEQ}$ - THE INSTITUTION FOR FIRST-ORDER PREDICATE LOGIC WITH EQUALITY

Based on Definition 28, we outline the components of the institution for first-order predicate logic with equality, $\mathcal{FOPEQ}$ [100].

Objects in the category of $\mathcal{FOPEQ}$-signatures are tuples of the form, $\Sigma_{\mathcal{FOPEQ}} = \langle S, \Omega, \Pi \rangle$, where S is a set of sort names, Ω is a set of

operation names indexed by arity and sort, and Π is a set of predicate names indexed by arity. Signature morphisms are sort/arity-preserving functions that rename sorts, operations and predicates.

For any $\Sigma_{\mathcal{FOPEQ}} = \langle S, \Omega, \Pi \rangle$, $\Sigma_{\mathcal{FOPEQ}}$-sentences are closed first-order formulae built out of atomic formulae using $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, \exists, \forall$. Atomic formulae are equalities between $\langle S, \Omega \rangle$-terms, predicate formulae of the form $p(t_1, \ldots, t_n)$ where $p \in \Pi$ and $t_1, \ldots, t_n$ are terms (with variables), and the logical constants `true` and `false`.

Given a signature $\Sigma_{\mathcal{FOPEQ}} = \langle S, \Omega, \Pi \rangle$, a model over $\mathcal{FOPEQ}$ consists of a carrier set $|A|_s$ for each sort name $s \in S$, a function $f_A : |A|_{s_1} \times \cdots \times |A|_{s_n} \rightarrow |A|_s$ for each operation name $f \in \Omega_{s_1 \ldots s_n, s}$ and a relation $p_A \subseteq |A|_{s_1} \times \cdots \times |A|_{s_n}$ for each predicate name $p \in \Pi_{s_1 \cdots s_n}$, where $s_1, \ldots, s_n$, and s are sort names.

The satisfaction relation in $\mathcal{FOPEQ}$ is the usual satisfaction of first-order sentences by first-order structures. This amounts to evaluating the truth of the $\mathcal{FOPEQ}$-formula using the values in the carrier sets supplied by the models.

2.3.4 REFINEMENT

As described in Chapter 1, the refinement calculus allows us to construct a specification gradually via a sequence of verifiable refinement steps [8, 86, 87]. In Section 2.1.1 we described how Event-B supports refinement, since refinement is so central to Event-B specification, any formalisation of the Event-B language must be capable of capturing refinement.

The theory of institutions equips us with a basic notion of refinement as *model-class inclusion* where the class of models of a specification con-

tains the models that satisfy the specification. We formally define what is meant by *model-class* in Definition 210.

Definition 210 (Model-Class)**.** For any $\Phi \subseteq \mathbf{Sen}(\Sigma)$ of Σ-sentences, the class $Mod_{\Sigma}(\Phi) \subseteq |\mathbf{Mod}(\Sigma)|$ of models of Φ is defined as the class of all Σ-models that satisfy all the sentences in Φ.

Note that we omit the Σ subscript where the signature is clear from the context. With regard to institution-theoretic refinement, the class of models of the concrete specification is a subset of the class of models of the abstract specification [100].

We consider two cases: (1) when the signatures are the same and (2) when the signatures are different. In the case where the signatures are the same, refinement is denoted as:

$$SP_A \sqsubseteq SP_C \quad \Leftrightarrow \quad Mod(SP_C) \subseteq Mod(SP_A)$$

where SP_A is an abstract specification that refines ($\sqsubseteq$) to a concrete specification SP_C, $Mod(SP_A)$ and $Mod(SP_C)$ denote the class of models of the abstract and concrete specifications respectively. This means that, given specifications with the same signature, the concrete specification should not exhibit any model that was not possible in the abstract specification. This equates to the definition of general refinement outlined in Chapter 1 [96].

In terms of Event-B, data refinement is not supported in this way because it would involve a change of signature. This was the case when refining the boolean variables in the traffic light example to variables over the COLOURS data type (Section 2.1.1). Instead, refinement occurs between machines by strengthening the invariants and the guards but does not provide any new variables or events. In this case, the specification is made more deterministic by further constraining the invariants and guards. This increase in determinism is supported by the refinement calculi described in Chapter 1 [8, 11, 86, 87].

When the signatures are different and related by a signature morphism $\sigma : Sig[SP_A] \rightarrow Sig[SP_C]$ then we can use the corresponding model morphism in order to express refinement. This model morphism is used to interpret the concrete specification as containing only the signature items from the abstract specification. Here, refinement is the model-class inclusion of the models of the concrete specification restricted into the class of models of the abstract specification using the model morphism. In this case we write:

$$SP_A \sqsubseteq SP_C \quad \Leftrightarrow \quad Mod(\sigma)(SP_C) \subseteq Mod(SP_A)$$

where $Mod(\sigma)(SP_C)$ is the model morphism applied to the model-class of the concrete specification SP_C. This interprets each of the models of SP_C as models of SP_A before a refinement relationship is determined.

In this scenario, both data refinement and superposition refinement are supported, since the refined or added variables and events can be removed, using a suitable model morphism as outlined above, before refinement is completed. Schneider et al. used a similar notion of hiding when they provided a CSP semantics for Event-B refinement [106].

In this section, we have defined what is meant by an institution and discussed institution-theoretic refinement with respect to Event-B. In Section 2.4, we describe how the theory of institutions can be used to address the limitations of Event-B that we identified in Section 2.2.

2.4 ADDRESSING THE LIMITATIONS OF EVENT-B

In Section 2.2, we identified two limitations of the Event-B formal specification language; (1) a lack of well-defined modularisation constructs and (2) an ad hoc approach to interoperability. Our thesis is that the theory of institutions can address these limitations and so we outline the potential solutions that they offer here.

Specification-Building Operator	Description
$SP1$ `with` σ	Renaming of signature components (sort, operation and predicate names in $\mathcal{FOPEQ}$ for example) using the signature morphism σ. The resultant specification is the same as the original with the appropriate renamings carried out. This corresponds to the functionality of the Renaming Refactory Rodin plugin described in Table 2.1.
$SP1$ `then` ...	Extends the specification $SP1$ by adding new sentences after the `then` specification-building operator. This operator can be used to represent superposition refinement of Event-B specifications by adding new variables and events.
$SP1$ `and` $SP2$	Combines the specifications $SP1$ and $SP2$. It is the most straightforward way of combining specifications with different signatures. This is achieved by forming a specification with a signature corresponding to the union of the signatures of $SP1$ and $SP2$.
$SP1$ `hide via` σ	Hiding via the signature morphism σ allows viewing a specification, $SP1$, as a specification containing only the signature components of another specified by the signature morphism σ. This is useful for representing refinement as it allows a concrete specification to be viewed as one written only using signature items supported by its corresponding abstract specification. Thus representing the refinement relation between specifications over different signatures.

Table 2.3: Institution-theoretic specification-building operators that can be used to modularise specifications in a formalism-independent manner. Note that $SP1$ and $SP2$ denote specifications written over some institution, and σ is a signature morphism in the same institution.

2.4.1 INSTITUTION-THEORETIC MODULARISATION CONSTRUCTS

The theory of institutions equips us with an array of specification-building operators that can be used to write modular specifications in a for-

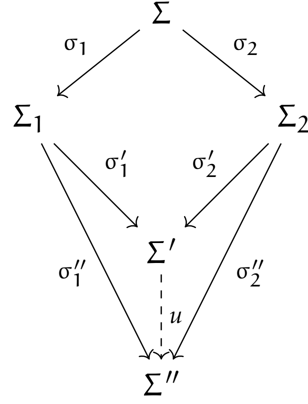
malism independent manner [15, 100]. The individual specification-building operators are summarised briefly in Table 2.3.

Built on these specification-building operators is the concept of *parametrised specifications* that are defined using other specifications as parameters. This is similar to the Generic Instantiation approach to Event-B modularisation described in Section 2.2.1. Parametrisation is a more general way of combining specifications than that of the specification-building operators providing λ-abstraction for user-defined abbreviations where variables in β-reduction range over specifications [100]. Parametrisation caters for a more elegant approach to modularisation than that achieved using specification-building operators in isolation.

We note that a parametrised specification can be rewritten in terms of the specification-building operators outlined in Table 2.3 [100]. The use of parametrisation in tandem with the `hide via` specification-building operator provides us with an elegant way to describe data refinement (an example of this can be viewed in Appendix A.2 [91].

A category-theoretic prerequisite for the successful usage of these specification-building operators and parametrisation, is that, for any given institution the category of signatures must have pushouts and the category of models must admit the amalgamation property [100].

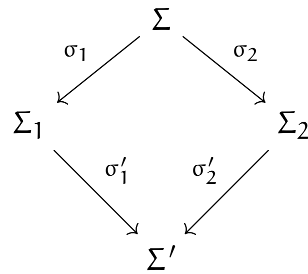
Definition 211 (Pushout). The pushout of two morphisms $\sigma_1 : \Sigma \to \Sigma_1$ and $\sigma_2 : \Sigma \to \Sigma_2$ is an object, Σ', together with a pair of morphisms $\sigma'_1 : \Sigma_1 \to \Sigma'$ and $\sigma'_2 : \Sigma_2 \to \Sigma'$ such that $\sigma'_1 \circ \sigma_1 = \sigma'_2 \circ \sigma_2$. For any other such $(\Sigma'', \sigma''_1, \sigma''_2)$, for which the following diagram commutes, there must be a unique $u : \Sigma' \to \Sigma''$ so that the following diagram commutes.



Pushouts provide a way to combine structures of various kinds. Using the diagram above, given objects Σ_1 and Σ_2, the pair of morphisms denoted by $\sigma_1 : \Sigma \to \Sigma_1$ and $\sigma_2 : \Sigma \to \Sigma_2$ indicate the common source that some "parts" of Σ_1 and Σ_2 come from. The pushout puts together Σ_1 and Σ_2 while identifying the parts coming from the common source and keeping the new parts disjoint [100].

For example in **Set**, the pushout is given by the disjoint union of Σ_1 and Σ_2, where elements that come from the same source (Σ) are identified, together with the morphisms σ'_1 and σ'_2 as shown above. Thus the pushout, P, in **Set**, is given by the formula $P = (\Sigma_1 \dot\cup \Sigma_2)/\sim$ where $\sim$ is the least equivalence relation such that $\sigma'_1 \circ \sigma_1(x) \sim \sigma'_2 \circ \sigma_2(x)$ for $x \in \Sigma$.

Definition 212 (Amalgamation). In any institution, a commuting square of signature morphisms



is an *amalgamation square* if and only if for each Σ_1-model M_1 and a Σ_2-model M_2 such that $M_1|_{\sigma_1} = M_2|_{\sigma_2}$, there exists a unique Σ'-model M' called the *amalgamation* of M_1 and M_2, such that $M'|_{\sigma'_1} = M_1$ and

$M'|_{\sigma_2'} = M_2$. The amalgamation M' may be denoted by $M_1 \otimes_{\sigma_1,\sigma_2} M_2$ or simply by $M_1 \otimes M_2$ [27].

In order for an institution to have good modularity properties, with respect to the specification-building operators, it is necessary that the institution have the *weak* amalgamation property where the uniqueness requirement of Definition 212 is dropped.

It is possible for the current approaches to modularisation for the Event-B language to be captured using these specification-building operators and a more comprehensive description of this on a per plugin basis can be found in Appendix A.

As mentioned earlier, the specification-building operators are generic in that they can be used in an institution independent manner. Thus, an interesting question is how specifications that are written using the specification-building operators in different institutions can be combined? We discuss interoperability between institutions in Section 2.4.2.

2.4.2 INSTITUTION-THEORETIC INTEROPERABILITY

In Section 2.2.2, we distinguished two kinds of interoperability for specification languages, that of translating from one language to another and that of heterogeneous specification. The theory of institutions supports both of these approaches by developing concepts of institution comorphisms (Definition 214), semi-morphisms (Definition 215), duplex institutions (Definition 216) and by using a Grothendieck construction [100].

In order to illustrate some of these constructs we define the institution of equational logic, $\mathcal{EQ}$ in Definition 213 [100].

Definition 213 ($\mathcal{EQ}$)**.** The institution of equational logic is composed of the following:

Objects of $\mathbf{Sign}_{\mathcal{EQ}}$ are pairs, $\langle S, \Omega \rangle$ where S is a set of sort names and Ω is a set of sort-indexed operation names.

Sentences in $\mathcal{EQ}$ are quantifier-free $\mathcal{FOPEQ}$ first-order logic formulae with equality as the only predicate.

Models consist of a carrier set $|A|_S$ for each sort name in S and a function $f_A : |A|_{s_1} \times |A|_{s_n} \to |A|_s$ for each operation name $f \in \Omega_{s_1 \ldots s_n, s}$.

The satisfaction relation is the usual satisfaction is the usual satisfaction of a Σ-equation by a Σ-model.

It is clear to see that this institution shares many commonalities with the institution for first-order logic, $\mathcal{FOPEQ}$, as described in Section 2.3.3 and we will illustrate some of these relationships in what follows.

Institution *comorphisms* embed a primitive institution into a more complex one and are defined as follows.

Definition 214 (**Institution Comorphism).** An institution comorphism $\rho : \mathbf{INS} \to \mathbf{INS}'$ is composed of:

A functor $\rho^{Sign} : \mathbf{Sign} \to \mathbf{Sign}'$.

A natural transformation $\rho^{Sen} : \mathbf{Sen} \to \rho^{Sign}; \ \mathbf{Sen}'$, that is, for each $\Sigma \in |\mathbf{Sign}|$, a function $\rho^{Sen}_{\Sigma} : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}'(\rho^{Sign}(\Sigma))$.

A natural transformation $\rho^{Mod} : (\rho^{Sign})^{op}; \ \mathbf{Mod}' \to \mathbf{Mod}$, that is, for each $\Sigma \in |\mathbf{Sign}|$, a functor $\rho^{Mod}_{\Sigma} : \mathbf{Mod}'(\rho^{Sign}(\Sigma)) \to \mathbf{Mod}(\Sigma)$.

An institution comorphism must ensure that for any signature $\Sigma \in |\mathbf{Sign}|$, the translations ρ^{Sen}_{Σ} of sentences and ρ^{Mod}_{Σ} of models preserve the satisfaction relation, that is, for any $\psi \in \mathbf{Sen}(\Sigma)$ and $M' \in |\mathbf{Mod}(\rho^{Sign}(\Sigma))|$:

$$\rho^{Mod}_{\Sigma}(M') \models_{\Sigma} \psi \quad \Leftrightarrow \quad M' \models'_{\rho^{Sign}(\Sigma)} \rho^{Sen}_{\Sigma}(\psi) \tag{2.2}$$

and the relevant diagrams, as indicated by Definition 27, in **Sen** and **Mod** commute for each signature morphism in **Sign** [100].

Example: It is clear that there is an institution comorphism $\rho : \mathcal{EQ} \to \mathcal{FOPEQ}$ that embeds $\mathcal{EQ}$ into $\mathcal{FOPEQ}$. It is structured as follows:

ρ^{Sign} : $\mathbf{Sign}_{\mathcal{EQ}} \to \mathbf{Sign}_{\mathcal{FOPEQ}}$ includes $\mathcal{EQ}$-signatures and their morphisms into the category of $\mathcal{FOPEQ}$-signatures by equipping them with the empty set of predicate symbols.

For each signature $\Sigma \in \mathbf{Sign}_{\mathcal{EQ}}$, $\rho^{Sen} : \mathbf{Sen}_{\mathcal{EQ}}(\Sigma) \to \mathbf{Sen}_{\mathcal{FOPEQ}}(\rho^{Sign}(\Sigma))$ maps any Σ-equation to the corresponding universally quantified equality as a $\mathcal{FOPEQ}$-sentence.

For each signature $\Sigma \in \mathbf{Sign}_{\mathcal{EQ}}$, $\rho^{Mod} : \mathbf{Mod}_{\mathcal{FOPEQ}}(\rho^{Sign}(\Sigma)) \to \mathbf{Mod}_{\mathcal{EQ}}(\Sigma)$ is the identity functor.

It is easy to see that the satisfaction condition holds.

Institution comorphisms supply the translation-based approach to interoperability. In Chapter 5 we will define a comorphism between our institution for Event-B (Chapter 3) [40, 41], and the institution for UML state machines [73]. This comorphism provides a mathematical foundation to the interoperability supplied by the UML-B Rodin plugin.

Tool support that offers interoperability between institutions is given by the Heterogeneous Toolset, HETS. HETS, written in Haskell, provides a general framework for parsing, static analysis and for proving the correctness of specifications in a formalism independent and thus heterogeneous manner [89]. In HETS, each formalism (expressed as an institution) is represented as a logic. In this setting, interoperability between formalisms is defined using institution comorphisms to relate the syntax of different logics and formalisms.

In contrast, institution *semi-morphisms* provide a means for constraining an institution by another and thus support a heterogeneous approach to interoperability although they are not supported in HETS.

Definition 215 (Institution Semi-Morphism). An institution semi-morphism $\mu : \mathbf{INS} \to \mathbf{INS'}$ consists of:

A functor $\mu^{Sign} : Sign \to Sign'$.

A natural transformation $\mu^{Mod} : Mod \to (\mu^{Sign})^{op}; Mod'$ [100].

Example: We can define an institution semi-morphism $\mu : \mathcal{FOPEQ} \to \mathcal{EQ}$ as containing the following components:

μ^{Sign} is the functor mapping any first-order signature $\Sigma = \langle S, \Omega, \Pi \rangle \in |\mathbf{Sign}_{\mathcal{FOPEQ}}|$ to the $\mathcal{EQ}$-signature $\mu^{Sign}(\Sigma) = \langle S, \Omega \rangle \in |\mathbf{Sign}_{\mathcal{EQ}}|$. The $\mathcal{FOPEQ}$-signature morphisms are mapped to $\mathcal{EQ}$-signature morphisms accordingly by forgetting about the predicate component of the morphisms.

For each $\mathcal{FOPEQ}$-signature, $\Sigma \in |\mathbf{Sign}_{\mathcal{FOPEQ}}|$, $\mu^{Mod} : \mathbf{Mod}_{\mathcal{FOPEQ}}(\Sigma) \to \mathbf{Mod}_{\mathcal{EQ}}(\mu^{Sign}(\Sigma))$ is the functor that forgets about the predicate component of the $\mathcal{FOPEQ}$-model and maps the model-morphisms accordingly [100].

Based on semi-morphisms as described in Definition 215, duplex institutions offer another approach to interoperability by enriching one institution by the sentences of another [100].

Definition 216 (Duplex Institutions). Given an institution semi-morphism $\mu : \mathbf{INS}_1 \to \mathbf{INS}_2$, we define the duplex institution $\mathbf{INS}_1$ `plus` $\mathbf{INS}_2$ `via` μ, which enriches $\mathbf{INS}_1$ by $\mathbf{INS}_2$-sentences reinterpreted by μ as follows:

$\mathbf{INS}_1$ `plus` $\mathbf{INS}_2$ `via` μ has the same signatures as $\mathbf{INS}_1$: $\mathbf{Sign} = \mathbf{Sign}_{\mathbf{INS}_1}$.

For each $\Sigma \in |\mathbf{Sign}|$, the set $\mathbf{Sen}(\Sigma)$ of Σ-sentences of $\mathbf{INS}_1$ `plus` $\mathbf{INS}_2$ `via` μ includes Σ-sentences of $\mathbf{INS}_1$ as well as $\mu^{Sign}(\Sigma)$-sentences of $\mathbf{INS}_2$, where the latter are written in the form ϕ_2 `via` μ, for $\phi_2 \in \mathbf{Sen}_{\mathbf{INS}_2}(\mu^{Sign}(\Sigma))$.

For each $\sigma : \Sigma \to \Sigma'$ in $\mathbf{Sign}$, $\mathbf{Sen}(\sigma) : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$ is defined as $\mathbf{Sen}_{\mathbf{INS}_1}(\sigma)$ on Σ-sentences in $\mathbf{Sen}_{\mathbf{INS}_1}(\Sigma) \subseteq \mathbf{Sen}(\Sigma)$, and

then for any ϕ_2 `via` $\mu \in \mathbf{Sen}(\Sigma)$, where $\phi_2 \in \mathbf{Sen}_{\mathbf{INS}_2}(\mu^{Sign}(\Sigma))$, $\mathbf{Sen}(\sigma)(\phi_2 \text{ via } \mu) = \mu^{Sign}(\sigma)(\phi_2)$.

$\mathbf{INS}_1$ `plus` $\mathbf{INS}_2$ `via` μ has the same models as $\mathbf{INS}_1$: $\mathbf{Mod} = \mathbf{Mod}_{\mathbf{INS}_1}$.

For each signature $\Sigma \in |\mathbf{Sign}|$, the satisfaction relation $\models_\Sigma$ is defined to coincide with $\models_{\mathbf{INS}_1,\Sigma}$ for Σ-sentences in $\mathbf{Sen}_{\mathbf{INS}_1}(\Sigma)$, while for $\phi_2 \in \mathbf{Sen}_{\mathbf{INS}_2}(\mu^{Sign}(\Sigma))$ and $M \in |\mathbf{Mod}(\Sigma)|$

$$M \models_\Sigma \phi_2 \texttt{via}\, \mu \Leftrightarrow \mu_\Sigma^{Mod}(M) \models_{\mathbf{INS}_2,\mu^{Sign}(\Sigma)} \phi_2$$

In this scenario, the sentences of $\mathbf{INS}_1$ and $\mathbf{INS}_2$ do not need to be related. Thus, reinterpreting the sentences of $\mathbf{INS}_2$ using this construction may increase the specification power of $\mathbf{INS}_1$ [100].

Example: Using $\mu : \mathcal{FOPEQ} \to \mathcal{EQ}$ as outlined above we can construct the duplex institution $\mathcal{FOPEQ}$ `plus` $\mathcal{EQ}$ `via` μ which allows equations to be used to specify properties of first-order structures [100]. This is not a very interesting construction as $\mathcal{EQ}$-sentences are already present in $\mathcal{FOPEQ}$. However, $\mathcal{EQ}$ might be equipped with powerful term rewriting and inductive reasoning tools which are not present in $\mathcal{FOPEQ}$ and so the duplex institution provides us with more expressivity than the institutions in isolation.

Institutions may also be combined (flattened) using the Grothendieck construction. This was the approach that was taken in CafeOBJ specification language (a successor of the OBJ specification language) [26, 28, 29]. The Grothendieck construction establishes a "disjoint sum" of a number of institutions and introduces theory morphisms across the institution embeddings.

2.5 SUMMARY

In this chapter we have examined the literature relevant to this thesis and introduced the mathematical prerequisites for our work. We will

refer to these definitions in later chapters and expand upon them where necessary.

The literature for Event-B indicates that a more unified approach is required for modular specifications and supporting interoperability between Event-B and other languages. It is our thesis that the theory of institutions is sufficient to provide the necessary constructs to support a unified approach, in a formally defined way. Our first step is to define an institution for Event-B as presented in Chapter 3.

Part I

DEFINING A SEMANTICS

"When you speak a new language you must see if you can translate all of the poetry of your old language into the new one."

– Dana Scott

3 DEFINING $\mathcal{EVT}$ - AN INSTITUTION FOR EVENT-B

In this chapter we present the formal definition of an institution for Event-B, called $\mathcal{EVT}$. This chapter provides our theoretical foundations for the work that we present in the chapters that follow. There are two basic languages within the Event-B language. The first one is the Event-B mathematical language (propositional/predicate logic, set-theory and arithmetic) and the second is the Event-B modelling language [3]. To represent the latter, we propose a new custom solution; for the former, however, we can use $\mathcal{FOPEQ}$, the institution of first-order predicate logic with equality that was presented in Section 2.3.3. Thus, our institution for Event-B is built on $\mathcal{FOPEQ}$. The work contained in this chapter was originally published in [40] and [41].

3.1 INTRODUCING $\mathcal{EVT}$

This chapter introduces the $\mathcal{EVT}$ institution and it is structured as follows. First we present $\mathbf{Sign}_{\mathcal{EVT}}$, $\mathbf{Sen}_{\mathcal{EVT}}$, $\mathbf{Mod}_{\mathcal{EVT}}$ and the satisfaction relation, $\models_{\mathcal{EVT}}$. We then prove that $\mathcal{EVT}$ preserves the institution-theoretic satisfaction condition, discuss the pragmatics of specification-building in $\mathcal{EVT}$ and define an institution comorphism from $\mathcal{FOPEQ}$ to $\mathcal{EVT}$. We show that $\mathcal{EVT}$ exhibits good behaviour with respect to modularisation by presenting pushouts and amalgamation in $\mathcal{EVT}$. We finish with an illustrative example of writing modular specifications in the $\mathcal{EVT}$ institution.

3.2 $\mathbf{Sign}_{\mathcal{EVT}}$, THE CATEGORY OF $\mathcal{EVT}$-SIGNATURES

Signatures describe the vocabulary that specifications written in a particular institution can have. Here, we define what it means to be an $\mathcal{EVT}$-signature (Definition 31) and an $\mathcal{EVT}$-signature morphism (Definition 32). We show that $\mathbf{Sign}_{\mathcal{EVT}}$ is indeed a category (Lemma 31) which is a necessary requirement for the definition of the $\mathcal{EVT}$ institution.

Definition 31 ($\mathcal{EVT}$**-Signature).** A **signature** in $\mathcal{EVT}$ is a five-tuple $\Sigma_{\mathcal{EVT}} = \langle S, \Omega, \Pi, E, V \rangle$ where $\langle S, \Omega, \Pi \rangle$ is a standard $\mathcal{FOPEQ}$-signature as outlined in Section 2.3.3, E is a set of events, i.e. of pairs $\langle \textit{event name}, \textit{status} \rangle$ where status belongs to the poset $\{\texttt{ordinary} < \texttt{anticipated} < \texttt{convergent}\}$, and V is a set of sorted variables. We assume that every signature has an initial event, called `Init`, whose status is always `ordinary`.

Notation: We write Σ in place of $\Sigma_{\mathcal{EVT}}$ when describing a signature over our institution for Event-B. For signatures over institutions other than $\mathcal{EVT}$ we will use the subscript notation where necessary; e.g. a signature over $\mathcal{FOPEQ}$ is denoted by $\Sigma_{\mathcal{FOPEQ}}$. For a given signature Σ, we access its individual components using a dot-notation, e.g. $\Sigma.V$ for the set V in the tuple Σ. We use the abbreviation `Init` to refer to the `Initialisation` event in Event-B.

Definition 32 ($\mathcal{EVT}$**-Signature Morphism).** A **signature morphism** $\sigma : \Sigma \to \Sigma'$ is a five-tuple containing σ_S, σ_Ω, σ_Π, σ_E and σ_V. Here σ_S, σ_Ω, σ_Π are the mappings taken from the corresponding signature morphism in $\mathcal{FOPEQ}$, as follows

- $\sigma_S : \Sigma.S \to \Sigma'.S$ is a function mapping sort names to sort names.
- $\sigma_\Omega : \Sigma.\Omega \to \Sigma'.\Omega$ is a family of functions mapping operation names in $\Sigma.\Omega$, respecting the arities and result sorts.
- $\sigma_\Pi : \Sigma.\Pi \to \Sigma'.\Pi$ is a family of functions mapping the predicate names in $\Sigma.\Pi$, respecting the arities.

and

- $\sigma_E : \Sigma.E \to \Sigma'.E$ is a function such that for any mapping $\sigma_E\langle e, st\rangle = \langle e', st'\rangle$ where $st \leqslant st'$; in addition, σ_E preserves the initial event: that is, $\sigma_E\langle \mathtt{Init}, \mathtt{ordinary}\rangle = \langle \mathtt{Init}, \mathtt{ordinary}\rangle$ and preserves the status ordering given by the poset $\{\mathtt{ordinary} < \mathtt{anticipated} < \mathtt{convergent}\}$.
- $\sigma_V : \Sigma.V \to \Sigma'.V$ is a sort-preserving function on sets of variable names. The set of variable names V contains elements of the form $v : s$ where v is the variable name and s is its sort. We apply a signature morphism σ to these elements as follows:

$$\sigma(v : s) = \sigma_V(v) : \sigma_S(s)$$

  where σ_V is a renaming function and σ_S is the sort mapping as described above.

Lemma 31. *$\mathcal{EVT}$-signatures and signature morphisms define a category* $\mathbf{Sign}_{\mathcal{EVT}}$*. The objects are signatures and the arrows are signature morphisms.*

Proof. Let $\Sigma = \langle S, \Omega, \Pi, E, V\rangle$ be an $\mathcal{EVT}$-signature where $\langle S, \Omega, \Pi\rangle$ is a signature over $\mathcal{FOPEQ}$, $\Sigma_{\mathcal{FOPEQ}}$, the institution for first-order predicate logic with equality [100]. E is a set of $\langle$*event name*, *status*$\rangle$ pairs and V is a set of sort-indexed variable names.

Recall from Definition 24 that, in a category, morphisms can be composed, are associative and identity morphisms exist. We show that $\mathbf{Sign}_{\mathcal{EVT}}$ preserves these three properties:

(a) Composition of $\mathcal{EVT}$-signature morphisms:
$\mathcal{EVT}$-signature morphisms can be composed, the composition of σ_S, σ_Ω and σ_Π is inherited from $\mathcal{FOPEQ}$. Therefore, we only examine the composition of σ_E and σ_V.

σ_E: Event names do not have a sort or arity, so the only restrictions are (1) on pairs of the form $\langle \mathtt{Init}, \mathtt{ordinary}\rangle$ and (2) that

the ordering in the status poset is preserved. In both of these cases the composition holds for σ_E.

σ_V: Variable names are sort-indexed so σ_V utilises σ_S on these sorts.

$$\begin{aligned}\sigma_2(\sigma_1(v:s)) &= \sigma_2((\sigma_{1_V}(v):\sigma_{1_S}(s))) \\ &= \sigma_{2_V}(\sigma_{1_V}(v)):\sigma_{2_S}(\sigma_{1_S}(s))\end{aligned}$$

Let $\sigma_1 : \Sigma_1 \to \Sigma_2$ and $\sigma_2 : \Sigma_2 \to \Sigma_3$, then we prove that $\sigma_2 \circ \sigma_1$ is a morphism in the category of $\mathcal{EVT}$-signatures.

– For all $\langle e_1, st_1\rangle \in \Sigma_1.E_1$, $\sigma_1(\langle e_1, st_1\rangle) \in \Sigma_2.E_2$ and for all $\langle e_2, st_2\rangle \in \Sigma_2.E_2$, $\sigma_2(\langle e_2, st_2\rangle) \in \Sigma_3.E_3$. Therefore $\sigma_2(\sigma_1(\langle e_1, st_1\rangle)) \in \Sigma_3.E_3$ so

$$\forall \langle e_1, st_1\rangle \in \Sigma_1.E_1 \Rightarrow \sigma_2 \circ \sigma_1(\langle e_1, st_1\rangle) \in \Sigma_3.E_3$$

– For all $(v_1 : s_1) \in \Sigma_1.V_1$, $\sigma_1((v_1 : s_1)) \in \Sigma_2.V_2$ and for all $(v_2 : s_2) \in \Sigma_2.V_2$, $\sigma_2((v_2 : s_2)) \in \Sigma_3.V_3$. Therefore $\sigma_2(\sigma_1((v_1 : s_1))) \in \Sigma_3.V_3$ so

$$\forall (v_1 : s_1) \in \Sigma_1.V_1 \Rightarrow \sigma_2 \circ \sigma_1((v_1 : s_1)) \in \Sigma_3.V_3$$

Therefore, $\mathcal{EVT}$-signature morphisms can be composed.

(b) Composition of $\mathcal{EVT}$-signature morphisms is associative:

$$(\sigma_3 \circ \sigma_2) \circ \sigma_1 = \sigma_3 \circ (\sigma_2 \circ \sigma_1)$$

For $\langle e, st\rangle \in \Sigma.E$:

$$\sigma_2 \circ \sigma_1(\langle e, st\rangle) = \sigma_2(\sigma_1(\langle e, st\rangle))$$

Then, by definition of composition

$$\begin{aligned}\sigma_3 \circ (\sigma_2 \circ \sigma_1)(\langle e, st\rangle) &= \sigma_3(\sigma_2(\sigma_1(\langle e, st\rangle))) \\ &= \sigma_3 \circ \sigma_2 \circ (\sigma_1(\langle e, st\rangle)) \\ &= (\sigma_3 \circ \sigma_2) \circ \sigma_1(\langle e, st\rangle)\end{aligned}$$

Similarly, for $(v : s) \in \Sigma.V$,

$$\sigma_3 \circ (\sigma_2 \circ \sigma_1)((v : s)) = (\sigma_3 \circ \sigma_2) \circ \sigma_1((v : s))$$

Therefore, the composition of $\mathcal{EVT}$-signature morphisms is associative.

(c) Identity morphisms for $\mathcal{EVT}$-signatures:
For any $\mathcal{EVT}$-signature Σ, there exists an identity signature morphism $id_\Sigma : \Sigma \to \Sigma$. id_E and id_V are such that $id_E(\langle e, st\rangle) = \langle e, st\rangle$ and $id_V((v : s)) = (v : s)$. This morphism satisfies the following signature morphism condition

$$\langle e, st\rangle \in \Sigma.E \Rightarrow id_E(\langle e, st\rangle) \in \Sigma.E \;\; \wedge \;\; (v : s) \in \Sigma.V \Rightarrow id_V(v : s) \in \Sigma.V$$

We have thus shown that $\mathbf{Sign}_{\mathcal{EVT}}$ forms a category as instructed by the definition of an institution (Definition 28).

□

In this section, we have defined the category of $\mathcal{EVT}$-signatures, $\mathbf{Sign}_{\mathcal{EVT}}$. Next, we define the functor $\mathbf{Sen}_{\mathcal{EVT}}$ that yields $\mathcal{EVT}$-sentences.

3.3 THE FUNCTOR $\mathbf{Sen}_{\mathcal{EVT}}$, YIELDING $\mathcal{EVT}$-SENTENCES

The second component of an institution is a functor called **Sen** that generates a set of sentences over a particular signature (Definition 28). Here, we define what is meant by a $\Sigma_{\mathcal{EVT}}$-Sentence (Definition 33 and we prove that $\mathbf{Sen}_{\mathcal{EVT}}$ is indeed a functor.

Definition 33 **($\Sigma_{\mathcal{EVT}}$-Sentence).** There are two kinds of sentence over $\mathcal{EVT}$:

$\langle e, \phi(\overline{x}, \overline{x\prime})\rangle$: This kind of sentence is used to represent individual events. Here, e is an event name in the domain of $\Sigma.E$ and $\phi(\overline{x}, \overline{x\prime})$ is an open $\mathcal{FOPEQ}$-formula over the variables $\overline{x}$ from $\Sigma.V$ and the primed versions, $\overline{x\prime}$, of the variables.

```
1  MACHINE m refines a SEES ctx
2    VARIABLES x̄
3    INVARIANTS I(x̄)                        ⟨inv, I(x̄) ∧ I(x̄′)⟩
4    VARIANT n(x̄)          {⟨e, n(x̄)′ ⩽ n(x̄)⟩, ⟨e, n(x̄)′ ⩽ n(x̄)⟩}
5    EVENTS
6    Initialisation ordinary
7      then  act-name: BA(x̄′)                 ⟨ Init, BA(x̄′)⟩
8    Event e ≙ status
9      any p̄
10     when  guard-name: G(x̄,p̄)
11     with  witness-name: W(x̄,p̄)
12     then  act-name: BA(x̄,p̄,x̄′)   ⟨e, ∃p̄ · G(x̄,p̄) ∧ W(x̄,p̄) ∧ BA(x̄,p̄,x̄′)⟩
       ⋮
13
14 END
```

Figure 3.1: The elements of an Event-B machine as presented in Rodin and their corresponding $\mathcal{EVT}$-sentences.

$\langle inv, \phi(\overline{x}, \overline{x}\prime)\rangle$ This kind of sentence is used to represent invariants. Here, *inv* is the tag used for sentences that define invariants and $\phi(\overline{x}, \overline{x}\prime)$ is as above.

In the Rodin Platform, Event-B machines are presented (syntactically sugared) as can be seen in Figure 3.1. The set of variables in a machine is denoted by $\overline{x}$ (line 2) and the invariants are denoted by $I(\overline{x})$ on line 3. The $\mathcal{EVT}$-signature derived from this machine would look as follows

$$\Sigma = \langle S, \Omega, \Pi, \{\langle \texttt{Initialisation}, \texttt{ordinary}\rangle, \langle \texttt{e}, \texttt{status}\rangle, \ldots\}, \{\overline{x}\}\rangle$$

where the $\mathcal{FOPEQ}$-component of the signature, $\langle S, \Omega, \Pi\rangle$, is drawn from the "seen" context(s) on line 1. The variant expression, denoted by $n(\overline{x})$ on line 4, is used for proving termination properties. As described in Section 2.1.1, events that have a status of `anticipated` or `convergent` must not increase and strictly decrease the variant expression respectively [3]. Each machine has an `Initialisation` event (lines 6–7) whose action is interpreted as a predicate $BA(\overline{x})$. Events can have parameter(s) as given by the list of identifiers $\overline{p}$ on line 9. $G(\overline{x}, \overline{p})$ and $W(\overline{x}, \overline{p})$ are predicates that represent the guard(s) and witness(es) respectively over the variables and parameter(s) (lines 10–11). Actions are interpreted as before-after predicates i.e. `x := x + 1` is interpreted as `x′ = x + 1`.

Thus, the predicate $BA(\overline{x}, \overline{p}, \overline{x}')$ on line 12 represents the action(s) over the parameter(s) $\overline{p}$ and the sets of variables $\overline{x}$ and $\overline{x}'$.

Formulae written in the mathematical language (such as the axioms that may appear in contexts) are interpreted as sentences over $\mathcal{FOPEQ}$. We can include these in specifications over $\mathcal{EVT}$ using the comorphism defined in Section 3.6. We represent the Event-B invariant, variant and event predicates as sentences over $\mathcal{EVT}$. These are illustrated alongside the Event-B predicates that they correspond to in Figure 3.1.

Invariants: For each Event-B invariant, $I(\overline{x})$, we form the open $\mathcal{FOPEQ}$-sentence $I(\overline{x}) \wedge I(\overline{x}\prime)$. Each invariant is paired with the *inv* invariant tag. Thus, we form the $\mathcal{EVT}$ invariant sentence $\langle inv, I(\overline{x}) \wedge I(\overline{x}\prime)\rangle$. When we define the $\mathcal{EVT}$-satisfaction relation, $\models_{\mathcal{EVT}}$, described Definition 37, we show how these sentences must hold for all events in the signature.

Variants: Firstly, we assume the existence of a suitable type for variant expressions and the usual interpretation of the predicates $<$ and $\leqslant$ over the integers in the signature. The variant expression applies to specific events, so we pair it with an event name in order to meaningfully evaluate it. This expression can be translated into an open $\mathcal{FOPEQ}$-term, which we denote by $n(\overline{x})$, and we use this to construct a formula based on the status of the event(s) in the signature Σ.

- For each $\langle e, \texttt{anticipated}\rangle \in \Sigma.E$ we form the $\mathcal{EVT}$-sentence

$$\langle e, n(\overline{x}\prime) \leqslant n(\overline{x})\rangle$$

- For each $\langle e, \texttt{convergent}\rangle \in \Sigma.E$ we form the $\mathcal{EVT}$-sentence

$$\langle e, n(\overline{x}\prime) < n(\overline{x})\rangle$$

Events: Event guard(s) and witnesses are also labelled predicates that can be translated into open $\mathcal{FOPEQ}$-formulae over the variables $\overline{x}$

```
1 ⟨ inv, cars_go ∈ BOOL ∧ cars_go′ ∈ BOOL ⟩
2 ⟨ inv, peds_go ∈ BOOL ∧ peds_go′ ∈ BOOL ⟩
3 ⟨ inv, ¬ (peds_go = true ∧ cars_go = true) ∧  ¬ (peds_go′ = true ∧ cars_go′ = true)⟩
4 ⟨ Init, cars_go′ = false ∧ peds_go′ = false ⟩
5 ⟨ set_peds_go, cars_go = false ∧ peds_go′ = true ⟩
6 ⟨ set_peds_stop, peds_go′ = false ⟩
7 ⟨ set_cars_go, peds_go = false ∧ cars_go′ = true ⟩
8 ⟨ set_cars_stop, cars_go′ = false ⟩
```

Figure 3.2: These are the $\mathcal{EVT}$-sentences corresponding to the abstract Event-B traffic light system as illustrated on lines 1–21 of Figure 2.1.

in V and parameters $\overline{p}$. These are denoted by $G(\overline{x},\overline{p})$ and $W(\overline{x},\overline{p})$ respectively. In Event-B, actions are interpreted as before-after predicates, thus they can be translated into open $\mathcal{FOPEQ}$-formulae denoted by $BA(\overline{x},\overline{p},\overline{x}\prime)$. Thus for each event we form the formula

$$\phi(\overline{x},\overline{x}\prime) = \exists\,\overline{p}\cdot G(\overline{x},\overline{p}) \wedge W(\overline{x},\overline{p}) \wedge BA(\overline{x},\overline{p},\overline{x}\prime)$$

where $\overline{p}$ are the event parameters. This generates an $\mathcal{EVT}$-sentence of the form $\langle e, \phi(\overline{x},\overline{x}\prime)\rangle$. The `Init` event, which is an Event-B sentence over only the after variables denoted by $\overline{x}\prime$, is a special case. In this case, we form the $\mathcal{EVT}$-sentence $\langle \texttt{Init}, \phi(\overline{x}\prime)\rangle$ where $\phi(\overline{x}\prime)$ is a predicate over the after values of the variables as assigned by the `Init` event.

Figure 3.2 contains the $\mathcal{EVT}$-sentences corresponding to the abstract Event-B machine on lines 1–21 of Figure 2.1.

Lemma 32. *There is a functor* $\mathbf{Sen}_{\mathcal{EVT}} : \mathbf{Sign}_{\mathcal{EVT}} \to \mathbf{Set}$ *generating for each $\mathcal{EVT}$-signature Σ a set of $\mathcal{EVT}$-sentences (objects in the category **Set**) and for each $\mathcal{EVT}$-signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ (morphisms in the category* $\mathbf{Sign}_{\mathcal{EVT}}$*) a function $Sen(\sigma) : Sen(\Sigma_1) \to Sen(\Sigma_2)$ (morphisms in the category **Set**) translating $\mathcal{EVT}$-sentences.*

Proof. $\mathbf{Sen}_{\mathcal{EVT}}$ is a functor therefore it is necessary to map the $\mathcal{EVT}$-signature morphisms to corresponding functions over sentences. The

functor maps morphisms to sentence morphisms respecting sort, arity and `Init` events. The image of a signature Σ_i ($i \in \mathbb{N}$) in $\mathbf{Sign}_{\mathcal{EVT}}$ is an object $\mathbf{Sen}(\Sigma_i)$ in the category **Set**. By the definition of the extension of the signature morphism to sentences [100], the translation of the object $\mathbf{Sen}(\Sigma_1)$ coincides with an object in $\mathbf{Sen}(\Sigma_2)$ with sort, operation, predicate, event and variable names translated with respect to the signature morphism $\sigma : \Sigma_1 \to \Sigma_2$. Thus, the image of a morphism in $\mathbf{Sign}_{\mathcal{EVT}}$ is a function $Sen(\sigma) : Sen(\Sigma_1) \to Sen(\Sigma_2)$ in the category **Set**.

Then we prove that **Sen** preserves the composition of $\mathcal{EVT}$-signature morphisms and identities as follows.

(a) Composition of $\mathcal{EVT}$-sentence morphisms:

$$Sen(\sigma_2 \circ \sigma_1) = Sen(\sigma_2) \circ Sen(\sigma_1)$$

Let Σ_i ($i = 1..4$) be $\mathcal{EVT}$-signatures and let $\sigma_1 : \Sigma_1 \to \Sigma_2$ and $\sigma_2 : \Sigma_3 \to \Sigma_4$ be $\mathcal{EVT}$-signature morphisms. Given an $\mathcal{EVT}$-sentence of the form $\langle e, \phi(\overline{x}, \overline{x\prime}) \rangle$, then by expanding each side of the following equivalence, since signature morphisms can be composed, we show that composition is preserved.

$$\begin{aligned} Sen(\sigma_2 \circ \sigma_1)(\langle e, \phi(\overline{x}, \overline{x\prime}) \rangle) &= Sen(\sigma_2) \circ Sen(\sigma_1)(\langle e, \phi(\overline{x}, \overline{x\prime}) \rangle) \\ \langle \sigma_2 \circ \sigma_1(e), \sigma_2 \circ \sigma_1(\phi(\overline{x}, \overline{x\prime})) \rangle &= Sen(\sigma_2)(\langle \sigma_1(e), \sigma_1(\phi(\overline{x}, \overline{x\prime})) \rangle) \\ \langle \sigma_2(\sigma_1(e)), \sigma_2(\sigma_1(\phi(\overline{x}, \overline{x\prime}))) \rangle &= \langle \sigma_2(\sigma_1(e)), \sigma_2(\sigma_1(\phi(\overline{x}, \overline{x\prime}))) \rangle \end{aligned}$$

Thus, composition is preserved.

(b) Preservation of identities:

Let $id_{\Sigma_1} : \Sigma_1 \to \Sigma_1$ be an identity $\mathcal{EVT}$-signature morphism as defined in Lemma 31. Since $\mathcal{EVT}$-signature morphisms already preserve identity and $Sen(id_{\Sigma_1})$ is the application of the identity signature morphisms to every element of the sentence, then the

identities are preserved. We can illustrate this as follows. Given a Σ_1-sentence $\langle e, \phi(\overline{x}, \overline{x\prime})\rangle$,

$$
\begin{aligned}
& Sen(id_{\Sigma_1}(\langle e, \phi(\overline{x}, \overline{x\prime})\rangle)) \\
& = \langle id_{\Sigma_1}(e), id_{\Sigma_1}(\phi(\overline{x}, \overline{x\prime}))\rangle \\
& = \langle e, \phi(\overline{x}, \overline{x\prime})\rangle
\end{aligned}
$$

and so, identities are preserved.

Thus $\mathbf{Sen}_{\mathcal{EVT}}$ is a functor. □

In this section, we have defined what is meant by an $\mathcal{EVT}$-sentence. Next, we define $\mathcal{EVT}$-models and the $\mathbf{Mod}_{\mathcal{EVT}}$ functor.

3.4 THE FUNCTOR $\mathbf{Mod}_{\mathcal{EVT}}$, YIELDING $\mathcal{EVT}$-MODELS

Our construction of $\mathcal{EVT}$-models is based on Event-B mathematical models as described by Abrial [3, Ch. 14]. In these models the state is represented as a sequence of variable-values and models are defined over before and after states. We interpret these states as sets of variable-to-value mappings in our definition of $\mathcal{EVT}$-models and so we define a Σ-state of an algebra A in Definition 34.

Definition 34 (Σ-***State***$_A$)**.** For any given $\mathcal{EVT}$-signature Σ we define a **Σ-state** of an algebra A as a set of (sort appropriate) variable-to-value mappings whose domain is the set of sort-indexed variable names $\Sigma.V$. We define the set $State_A$ as the set of all such Σ-states. By "sort appropriate" we mean that for any variable x of sort s in V, the corresponding value for x should be drawn from $|A|_s$, the carrier set of s given by a $\mathcal{FOPEQ}$-model A.

Definition 35 ($\Sigma_{\mathcal{EVT}}$**-Model**)**.** Given $\Sigma = \langle S, \Omega, \Pi, E, V\rangle$, $\mathbf{Mod}_{\mathcal{EVT}}(\Sigma)$ provides a category of $\mathcal{EVT}$-models, where an $\mathcal{EVT}$-**model** over Σ is a tuple $\langle A, L, R\rangle$ where

```
1  Event e ≙
2    when  grd1:    x<2
3    then  act1:    x := x + 1
4          act2:    y := false
```

Figure 3.3: An example of an Event-B event, e, with natural number variable x and boolean variable y. When $x > 2$, the event increments the value of x and toggles y to `false`.

- A is a $\Sigma_{\mathcal{FOPEQ}}$-model.
- $L \subseteq State_A$ is the non-empty initialising set that provides the states after the `Init` event. We note that trivial models are excluded as the initialising set L is never empty. We can see this because even in the extreme cases where there are no predicates or variables in the `Init` event, L is the singleton containing the empty map ($L = \{\{\}\}$).
- For every event name $e \in dom(E)$, other than `Init`, we define $R.e \subseteq State_A \times State_A$ where for each pair of states $\langle s, s\prime \rangle$ in $R.e$, s provides values for the variables x in V, and $s\prime$ provides values for their primed versions $x\prime$. Then $R = \{R.e \mid e \in dom(E) \textit{ and } e \neq \texttt{Init}\}$.

Intuitively, a model over Σ maps every event name e in $(\Sigma.E)$ to a set of variable-to-value mappings over the carriers corresponding to the sorts of each of the variables $x \in \Sigma.V$ and their primed versions $x\prime$.

For example, given the event `e` in Figure 3.3, with natural number variable `x` and boolean variable `y` we construct the variable-to-value mappings:

$$R_e = \left\{ \begin{array}{c} \{x \mapsto 0, y \mapsto \mathit{false}, x\prime \mapsto 1, y\prime \mapsto \mathit{false}\}, \\ \{x \mapsto 0, y \mapsto \mathit{true}, x\prime \mapsto 1, y\prime \mapsto \mathit{false}\}, \\ \{x \mapsto 1, y \mapsto \mathit{false}, x\prime \mapsto 2, y\prime \mapsto \mathit{false}\}, \\ \{x \mapsto 1, y \mapsto \mathit{true}, x\prime \mapsto 2, y\prime \mapsto \mathit{false}\} \end{array} \right\}$$

The notation used in this example is interpreted as *variable name* $\mapsto$ *value* which is a data state where the value is drawn from the carrier set corresponding to the sort of the variable name given in $\Sigma.V$.

Thus we have defined what is meant by an $\mathcal{EVT}$-model.

In Lemma 33, we prove that $\mathbf{Mod}_{\mathcal{EVT}}(\Sigma)$ forms a category for a given $\mathcal{EVT}$-signature Σ.

Lemma 33. *For any $\mathcal{EVT}$-signature Σ there is a category of $\mathcal{EVT}$-models $\mathbf{Mod}_{\mathcal{EVT}}(\Sigma)$ where the objects in the category are $\mathcal{EVT}$-models and the arrows are $\mathcal{EVT}$-model morphisms.*

Proof. We begin by describing $\mathcal{EVT}$-model morphisms and then prove that composition and identities are preserved.

In $\mathcal{FOPEQ}$ a model morphism $h : A_1 \to A_2$ is a family of functions $h = \langle h_s : |A_1|_s \to |A_2|_s \rangle_{s \in S}$ which respects the sorts and arities of the operations and predicates. Recall from Definition 35 that $\mathcal{EVT}$-models have the form $\langle A, L, R \rangle$, therefore $\mathcal{EVT}$-model morphisms are given by extending the corresponding $\mathcal{FOPEQ}$-model morphisms for the A component of the model to the initialising set L and the relations in R.

Thus for each $\mathcal{EVT}$-model morphism $\mu : \langle A_1, L_1, R_1 \rangle \to \langle A_2, L_2, R_2 \rangle$ there is an underlying $\mathcal{FOPEQ}$-model morphism $h : A_1 \to A_2$, and we extend this to the states in the set L_1 and in the relation R_1. That is, for any element

$$\{x_1 \mapsto a_1, ..., x_n \mapsto a_n, x_1\prime \mapsto a_1\prime, ..., x_n\prime \mapsto a_n\prime\} \in R_1.e$$

in R_1 there is

$$\{x_1 \mapsto h(a_1), ..., x_n \mapsto h(a_n), x_1\prime \mapsto h(a_1\prime), ..., x_n\prime \mapsto h(a_n\prime)\} \in R_2.e$$

in R_2 where $x_1, \ldots, x_n, x_1\prime, \ldots, x_n\prime$ are variable names and their primed versions drawn from V. A similar construction follows for $L1$. The composition of model morphisms, their associativity and identity derives from that of $\mathcal{FOPEQ}$.

(a) Composition of $\mathcal{EVT}$-model morphisms:
Let $M_i = \langle A_i, L_i, R_i \rangle$ be a model and $h_i : M_i \to M_{i+1}$ be an $\mathcal{EVT}$-model morphism where $i \in \mathbb{N}$. We can now show that the composition of $\mathcal{EVT}$-model morphisms is associative as follows:

$$\begin{aligned}
(h_3 \circ h_2) \circ h_1 &= h_3 \circ (h_2 \circ h_1) \\
(h_3 \circ h_2)(h_1(M_1)) &= h_3 \circ (h_2(h_1(M_1))) \\
(h_3 \circ h_2)(M_2) &= h_3 \circ (h_2(M_2)) \\
h_3(h_2(M_2)) &= h_3(h_2(M_2)) \\
h_3(M_3) &= h_3(M_3) \\
M_4 &= M_4
\end{aligned}$$

(b) Identity morphism for $\mathcal{EVT}$-models:
For any $\mathcal{EVT}$-model M_i there exists an identity model morphism $h_{id} : M_i \to M_i$. If $M_i = \langle A_i, L_i, R_i \rangle$ then $h_{id}(M_i) = \langle A_i, L_i, R_i \rangle$

Thus $\mathbf{Mod}_{\mathcal{EVT}}(\Sigma)$ forms a category. □

Model reducts are central to the definition of an institution as outlined in Section 2.3.2 and in Definition 36, we define the $\mathcal{EVT}$-model reduct. Then, in Lemma 34, we prove that the $\mathcal{EVT}$-model reduct is a functor.

Definition 36 (**$\mathcal{EVT}$-model reduct**). The reduct of an $\mathcal{EVT}$-model $M = \langle A, L, R \rangle$ along an $\mathcal{EVT}$-signature morphism $\sigma : \Sigma \to \Sigma'$ is given by $M|_\sigma = \langle A|_\sigma, L|_\sigma, R|_\sigma \rangle$. Here $A|_\sigma$ is the reduct of the $\mathcal{FOPEQ}$-component of the $\mathcal{EVT}$-model along the $\mathcal{FOPEQ}$-components of σ. $L|_\sigma$ and $R|_\sigma$ are based on the reduction of the states of A along σ, i.e. for every Σ'-state s of A, that is for every sorted map $s : \Sigma'.V \to |A|$, $s|_\sigma$ is the map $\Sigma'.V \to |A|$ given by the composition $\sigma_V; s$. This extends in the usual manner from states to sets of states and to relations on states.

Lemma 34. *For each $\mathcal{EVT}$-signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ the $\mathcal{EVT}$-model reduct is a functor **Mod**(σ) from Σ_2-models to Σ_1-models.*

Recall that each Σ-State$_A$ is a set of variable-to-value mappings of the form

$$\{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$$

where $x_1, \ldots, x_n \in \Sigma.V$ (Definition 34).

Proof. Let $M_2 = \langle A_2, L_2, R_2 \rangle$ be a Σ_2-model. Then the reduct $M_2|_\sigma$ collapses the $\mathcal{EVT}$-model to only contain signature items supported by Σ_1 and consists of the tuple $M_2|_\sigma = \langle A_2|_\sigma, L_2|_\sigma, R_2|_\sigma \rangle$ such that

- $A_2|_\sigma$ is the reduct of the $\mathcal{FOPEQ}$-component of the $\mathcal{EVT}$-model along the $\mathcal{FOPEQ}$-components of $\sigma : \Sigma \to \Sigma'$.
- $L_2|_\sigma$ and $R_2|_\sigma$ are based on the reduction of the states of A_2 along σ. In particular, given $e \in dom(E1)$ and $e \neq \texttt{Init}$ and $R_2.\sigma(e) \in R_2$

  $$R_2.\sigma(e) = \{s_1, ..., s_m\}$$

  where each s_i is a Σ_2-state$_{A_2}$ $(1 \leqslant i \leqslant m)$ is of the form

  $$\{\sigma(x_1) \mapsto a_1, ..., \sigma(x_n) \mapsto a_n, \sigma(x_1\prime) \mapsto a_1\prime, ..., \sigma(x_n\prime) \mapsto a_n\prime\}$$

  with $x_1, ..., x_n \in \Sigma.V$ and $x_1\prime, ..., x_n\prime \in \Sigma.V\prime$.

  Then for each $e \in dom(E1)$, $e \neq \texttt{Init}$ and $R_2|_\sigma.e \in R_2|_\sigma$ there is

  $$R_2|_\sigma.e = \{s_1|_\sigma, ..., s_m|_\sigma\}$$

  where each $s_i|_\sigma$ $(1 \leqslant i \leqslant m)$ is of the form

  $$\{x_1 \mapsto a_1, ..., x_n \mapsto a_n, x_1\prime \mapsto a_1\prime, ..., x_n\prime \mapsto a_n\prime\}$$

In order to prove that the model reduct is a functor, we show that it preserves composition and identites as follows:

(a) Preservation of composition for $\mathcal{EVT}$-model reducts:
Our objective here is to show that the reduct of a composition of two $\mathcal{EVT}$-model morphisms is equal to the composition of the

redicts of those $\mathcal{EVT}$-model morphisms. Given $\mathcal{EVT}$-model morphisms $h_1 : M_1 \to M_2$ and $h_2 : M_2 \to M_3$, then we show that

$$(h_2 \circ h_1)|_\sigma = h_2|_\sigma \circ h_1|_\sigma$$

for some $\mathcal{EVT}$-signature morphism $\sigma : \Sigma \to \Sigma'$. Given an $\mathcal{EVT}$-model of the form $M_1 = \langle A, L, R \rangle$ over Σ, then for any $R.e \in R$, as outlined above, of the form

$$\{x_1 \mapsto a_1, \ldots, x\prime_n \mapsto a\prime_n\}_e$$

Then $(h_2 \circ h_1)\ |_\sigma$ is defined as

$$(h_2 \circ h_1)\{\sigma(x_1) \mapsto a_1, \ldots, \sigma(x\prime_n) \mapsto a\prime_n\}_{\sigma}(e)$$

This is equal to

$$h_2(\{\sigma(x_1) \mapsto h_1(a_1), \ldots, \sigma(x\prime_n) \mapsto h_1(a\prime_n)\}_{\sigma}(e)$$

Since $\mathcal{EVT}$-model morphisms can be composed, this is thus equal to $h_2|_\sigma \circ h_1|_\sigma$.

(b) Preservation of identities for $\mathcal{EVT}$-model reducts:
The reduct of the identity is the identity. Let id_{M_2} be an identity Σ_2-morphism then $id_{M_2}|_\sigma$ is an identity Σ_1-morphism h_1 defined by $h_1(R.e) = id_{M_2}|_\sigma(R.e) = R.e$ for any $R.e \in R$ and e is an event other than `Init`.

For the components belonging to A these proofs follow the corresponding proofs in $\mathcal{FOPEQ}$. □

Lemma 35. *There is a functor* $\mathbf{Mod}_{\mathcal{EVT}}$ *yielding a category* $\mathbf{Mod}(\Sigma)$ *of* $\mathcal{EVT}$-*models for each* $\mathcal{EVT}$-*signature* Σ, *and for each* $\mathcal{EVT}$-*signature morphism* $\sigma : \Sigma_1 \to \Sigma_2$ *a functor* $\mathbf{Mod}(\sigma)$ *from* Σ_2-*models to* Σ_1-*models.*

Proof. For each $\sigma : \Sigma_1 \to \Sigma_2$ in $\mathbf{Sign}_{\mathcal{EVT}}$ there is an arrow in $\mathbf{Sign}^{op}_{\mathcal{EVT}}$ going in the opposite direction. By Lemma 4, the image of this arrow

in $\mathbf{Sign}^{op}_{\mathcal{EVT}}$ is $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma_2) \to \mathbf{Mod}(\Sigma_1)$ in **Cat**. By Lemma 3, the image of a signature in $\mathbf{Sign}_{\mathcal{EVT}}$ is an object $\mathbf{Mod}(\Sigma)$ in **Cat**. Therefore, domain and codomain of the image of an arrow are the images of the domain and codomain respectively.

(a) Preservation of composition:

$$\mathbf{Mod}(\sigma_2 \circ \sigma_1) = \mathbf{Mod}(\sigma_2) \circ \mathbf{Mod}(\sigma_1)$$

Let $\sigma_1 : \Sigma_1 \to \Sigma_2$ and $\sigma_2 : \Sigma_2 \to \Sigma_3$ be $\mathcal{EVT}$-signature morphisms and let $M_i = \langle A_i, L_i, R_i \rangle$ be an $\mathcal{EVT}$-model over Σ_i and let h_i be a Σ_i-model morphism with $i \in \{1, 2, 3\}$.

- $M_3|_{\sigma_2 \circ \sigma_1} = (M_3|_{\sigma_2})|_{\sigma_1}$
  By definition of reduct
  $M_3|_{\sigma_2} = \langle A_3, L_3, R_3 \rangle|_{\sigma_2} = \langle A_2, L_2, R_2 \rangle = M_2$.
  Then
  $(M_3|_{\sigma_2})|_{\sigma_1} = M_2|_{\sigma_1} = \langle A_2, L_2, R_2 \rangle|_{\sigma_1} = \langle A_1, L_1, R_1 \rangle = M_1$.
  By composition of signature morphisms $\sigma_2 \circ \sigma_1 : \Sigma_1 \to \Sigma_3$, then

  $$M_3|_{\sigma_2 \circ \sigma_1} = \langle A_3, L_3, R_3 \rangle|_{\sigma_2 \circ \sigma_1} = \langle A_1, L_1, R_1 \rangle = M_1$$

  Therefore $M_3|_{\sigma_2 \circ \sigma_1} = (M_3|_{\sigma_2})|_{\sigma_1}$
- $h_3|_{\sigma_2 \circ \sigma_1} = (h_3|_{\sigma_2})|_{\sigma_1}$
  Proof similar to above.

(b) Preservation of identities:
Let id_{Σ_1} be an identity signature morphism as defined in Lemma 31. Since $\mathcal{EVT}$-signature morphisms already preserve identity and $Mod(id_{\Sigma_1})$ is the application of the identity signature morphisms to every part of the $\mathcal{EVT}$-model, the identities are preserved.

□

In this section, we defined $\mathcal{EVT}$-models, model reducts and the $\mathbf{Mod}_{\mathcal{EVT}}$ functor. Next, we describe the satisfaction relation, $\models_{\mathcal{EVT}}$, in $\mathcal{EVT}$ and prove that $\mathcal{EVT}$ is a valid institution.

3.5 THE SATISFACTION RELATION FOR $\mathcal{EVT}$

All institutions are equipped with a satisfation relation to evaluate if a particular model satisfies a sentence. Here, we define the satisfaction relation for $\mathcal{EVT}$. In order to define the satisfaction relation for $\mathcal{EVT}$, we describe an embedding from $\mathcal{EVT}$-signatures and models to $\mathcal{FOPEQ}$-signatures and models.

Given an $\mathcal{EVT}$-signature $\Sigma = \langle S, \Omega, \Pi, E, V \rangle$ we form the following two $\mathcal{FOPEQ}$-signatures:

- $\Sigma^{(V,V\prime)}_{\mathcal{FOPEQ}} = \langle S, \Omega \cup V \cup V\prime, \Pi \rangle$ where V and $V\prime$ are the variables and their primed versions, respectively, that are drawn from the $\mathcal{EVT}$-signature, and represented as 0-ary operators with unchanged sort. The intuition here is that the set of variable-to-value mappings for the free variables in an $\mathcal{EVT}$-signature Σ are represented by adding a distinguished 0-ary operation symbol to the corresponding $\mathcal{FOPEQ}$-signature for each of the variables $x \in V$ and their primed versions.
- Similarly, for the initial state and its variables, we construct the signature $\Sigma^{(V\prime)}_{\mathcal{FOPEQ}} = \langle S, \Omega \cup V\prime, \Pi \rangle$.

Given the $\mathcal{EVT}$ Σ-model $\langle A, L, R \rangle$, we construct the $\mathcal{FOPEQ}$-models:

- For every pair of states $\langle s, s\prime \rangle$, we form the $\Sigma^{(V,V\prime)}_{\mathcal{FOPEQ}}$-model expansion $A^{(s,s\prime)}$, which is the $\mathcal{FOPEQ}$-component A of the $\mathcal{EVT}$-model, with s and $s\prime$ added as interpretations for the new operators that correspond to the variables from V and $V\prime$ respectively.
- For each initial state $s\prime \in L$ we construct the $\Sigma^{(V\prime)}_{\mathcal{FOPEQ}}$-model expansion $A^{(s\prime)}$ analogously.

For any $\mathcal{EVT}$-sentence over Σ of the form $\langle e, \phi(\overline{x}, \overline{x\prime})\rangle$ or $\langle inv, \phi(\overline{x}, \overline{x\prime})\rangle$, we create a corresponding $\mathcal{FOPEQ}$-formula by replacing the free variables with their corresponding operator symbols. We write this (closed) formula as $\phi(\overline{x}, \overline{x}')$.

Given these $\mathcal{FOPEQ}$-signatures and models, we now define the satisfaction relation for $\mathcal{EVT}$ in Definition 37.

Definition 37 (Satisfaction Relation). Since there are two kinds of $\mathcal{EVT}$-sentence, we define two kinds of satisfaction relation in $\mathcal{EVT}$.

Satisfaction Relation 1: For any $\mathcal{EVT}$-model $\langle A, L, R\rangle$ and $\mathcal{EVT}$-sentence $\langle e, \phi(\overline{x}, \overline{x\prime})\rangle$, where e is an event name other than `Init`, we define:

$$\langle A, L, R\rangle \models_{\Sigma} \langle e, \phi(\overline{x}, \overline{x\prime})\rangle \quad \Leftrightarrow \quad \forall \langle s, s\prime\rangle \in R.e \cdot A^{(s,s\prime)} \models_{\Sigma^{(V,V\prime)}_{\mathcal{FOPEQ}}} \phi(\overline{x}, \overline{x}')$$

Similarly, we evaluate the satisfaction relation of $\mathcal{EVT}$-sentences of the form $\langle \texttt{Init}, \phi(\overline{x\prime})\rangle$ as follows:

$$\langle A, L, R\rangle \models_{\Sigma} \langle \texttt{Init}, \phi(\overline{x\prime})\rangle \quad \Leftrightarrow \quad \forall s\prime \in L \cdot A^{(s\prime)} \models_{\Sigma^{(V\prime)}_{\mathcal{FOPEQ}}} \phi(\overline{x}')$$

Satisfaction Relation 2: For any $\mathcal{EVT}$-model $\langle A, L, R\rangle$ and $\mathcal{EVT}$-sentence $\langle inv, \phi(\overline{x}, \overline{x\prime})\rangle$, we define:

$$\langle A, L, R\rangle \models_{\Sigma} \langle inv, \phi(\overline{x}, \overline{x\prime})\rangle$$

$$\Leftrightarrow$$

For every non-`Init` event, e,

$$\langle \forall s, s\prime\rangle \in R.e \cdot A^{(s,s\prime)} \models_{\Sigma^{(V,V\prime)}_{\mathcal{FOPEQ}}} \phi(\overline{x}, \overline{x}')$$

$$\wedge \quad \forall s\prime \in L \cdot A^{(s\prime)} \models_{\Sigma^{(V\prime)}_{\mathcal{FOPEQ}}} \phi(\overline{x}, \overline{x}')$$

Theorem 31 (Satisfaction Condition). *Given $\mathcal{EVT}$ signatures Σ_1 and Σ_2, a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, a Σ_2-model M_2 and a Σ_1-sentence ψ_1, the following satisfaction condition holds:*

$$\mathbf{Mod}(\sigma)(M_2) \models_{\mathcal{EVT}_{\Sigma_1}} \psi_1 \quad \Leftrightarrow \quad M_2 \models_{\mathcal{EVT}_{\Sigma_2}} \mathbf{Sen}(\sigma)(\psi_1)$$

Proof. We must prove this for both kinds of $\mathcal{EVT}$-sentence ($\langle e, \phi(\overline{x}, \overline{x}\prime)\rangle$ and $\langle inv, \phi(\overline{x}, \overline{x}\prime)\rangle$) as defined in Defintion 33, with the satisfaction relation, $\models_{\mathcal{EVT}}$, as defined in Definition 37. Let M_2 be the $\mathcal{EVT}$-model $\langle A_2, L_2, R_2\rangle$, then,

- Let ψ_1 be the sentence $\langle e, \phi(\overline{x}, \overline{x}\prime)\rangle$, then the satisfaction condition is equivalent to

$$\forall\langle s, s\prime\rangle \in R_2|_\sigma.e \cdot (A_2|_\sigma)^{(s,s\prime)}|_\sigma \models_{\mathcal{FOPEQ}_{\Sigma^{(V_1,V_1\prime)}_{\mathcal{FOPEQ}}}} \phi(\overline{x}, \overline{x}')$$
$$\Leftrightarrow \forall\langle s, s\prime\rangle \in R_2.\sigma_E(e) \cdot A_2^{(s,s\prime)} \models_{\mathcal{FOPEQ}_{\Sigma^{(V_2,V_2\prime)}_{\mathcal{FOPEQ}}}} Sen(\sigma)(\phi(\overline{x}, \overline{x}'))$$

  Here, validity follows from the validity of satisfaction in $\mathcal{FOPEQ}$. We prove the result for initial events in the same way.

- Let ψ_1 the sentence $\langle inv, \phi(\overline{x}, \overline{x}\prime)\rangle$, then the satisfaction condition is equivalent to

$$\begin{aligned}
&\forall e \in dom(\Sigma_2 \mid_\sigma .E) \wedge e \neq \mathtt{Init} \Rightarrow \\
&\quad \forall\langle s, s\prime\rangle \in R_2 \mid_\sigma .e \cdot (A_2 \mid_\sigma)^{(s,s\prime)} \mid_\sigma \models_{\mathcal{FOPEQ}_{\Sigma^{(V_1,V_1\prime)}_{\mathcal{FOPEQ}}}} \phi(\overline{x}, \overline{x}') \\
&\quad \wedge \quad \forall s\prime \in L_2|_\sigma \cdot (A_2|_\sigma)^{(s\prime)}|_\sigma) \models_{\Sigma^{(V_1\prime)}_{\mathcal{FOPEQ}}} \phi(\overline{x}, \overline{x}') \\
&\Leftrightarrow \\
&\quad \forall\, \sigma(e) \in dom(\Sigma_2.E) \wedge e \neq \mathtt{Init} \Rightarrow \\
&\quad \forall\langle s, s\prime\rangle \in R_2.\sigma_E(e) \cdot A_2^{(s,s\prime)} \models_{\mathcal{FOPEQ}_{\Sigma^{(V_2,V_2\prime)}_{\mathcal{FOPEQ}}}} \mathbf{Sen}(\sigma)(\phi(\overline{x}, \overline{x}')) \\
&\quad \wedge \quad \forall s\prime \in L_2 \cdot A_2^{(s\prime)} \models_{\Sigma^{(V_2\prime)}_{\mathcal{FOPEQ}}} \mathbf{Sen}(\sigma)(\phi(\overline{x}, \overline{x}'))
\end{aligned}$$

  Again, the validity follows from the validity of satisfaction in $\mathcal{FOPEQ}$.

In both cases, validity follows from that of $\mathcal{FOPEQ}$ and thus $\mathcal{EVT}$ preserves the satisfaction condition required of an institution.

□

In this section we have presented the satisfaction relation in $\mathcal{EVT}$ and showed that $\mathcal{EVT}$ preserves the axiomatic property of an institution (satisfaction condition in Theorem 31). As the satisfaction relation in $\mathcal{EVT}$ (Definition 37) relied on a $\mathcal{FOPEQ}$ embedding of signatures and models, we discuss the relationship between $\mathcal{FOPEQ}$ and $\mathcal{EVT}$ in Section 3.6.

3.6 RELATING $\mathcal{FOPEQ}$ AND $\mathcal{EVT}$

Initially, we defined the relationship between $\mathcal{FOPEQ}$ and $\mathcal{EVT}$ to be a duplex institution formed from a restricted version of $\mathcal{EVT}$ ($\mathcal{EVT}_{res}$) and $\mathcal{FOPEQ}$ where $\mathcal{EVT}_{res}$ is the institution $\mathcal{EVT}$ but does not contain any $\mathcal{FOPEQ}$ components. As presented in Section 2.4.2 (Definition 216), duplex institutions are constructed by enriching one institution by the sentences of another ($\mathcal{EVT}_{res}$ is enriched by sentences from $\mathcal{FOPEQ}$) using an institution semi-morphism [52, 100]. This approach would allow us to constrain $\mathcal{EVT}_{res}$ by $\mathcal{FOPEQ}$ and thus facilitate the use of $\mathcal{FOPEQ}$-sentences in an elegant way. However, duplex institutions are not supported in HETS [89], and therefore we opt for a comorphism (Definition 214) which embeds the simpler institution $\mathcal{FOPEQ}$ into the more complex institution $\mathcal{EVT}$ [100].

Definition 38 (The institution comorphism $\rho : \mathcal{FOPEQ} \to \mathcal{EVT}$ **).** We define $\rho : \mathcal{FOPEQ} \to \mathcal{EVT}$ to be an institution comorphism composed of:

- The functor $\rho^{Sign} : \mathbf{Sign}_{\mathcal{FOPEQ}} \to \mathbf{Sign}_{\mathcal{EVT}}$ which takes as input a $\mathcal{FOPEQ}$-signature of the form $\langle S, \Omega, \Pi \rangle$ and extends it with the set $E = \{\langle \texttt{Init ordinary} \rangle\}$ and an empty set of variable names V. $\rho^{Sign}(\sigma)$ works as σ on S, Ω and Π, it is the identity on the `Init` event and the empty function on the empty set of variable names.
- The natural transformation $\rho^{Sen} : \mathbf{Sen}_{\mathcal{FOPEQ}} \to \rho^{Sign}; \mathbf{Sen}_{\mathcal{EVT}}$ which pairs any closed $\mathcal{FOPEQ}$-sentence (given by ϕ) with the invariant

sentence identifier, *inv*, to form the $\mathcal{EVT}$-sentence $\langle inv, \phi\rangle$. As there are no variables in the signature, and ϕ is any closed formula, we do not require ϕ to be over the variables $\overline{x}$ and $\overline{x\prime}$.

- The natural transformation $\rho^{Mod} : (\rho^{Sign})^{op}; \ \mathbf{Mod}_{\mathcal{EVT}} \to \mathbf{Mod}_{\mathcal{FOPEQ}}$ is such that for any $\mathcal{FOPEQ}$-signature Σ,
$$\rho^{Mod}_{\Sigma}(Mod(\rho^{Sign}(\Sigma))) = \rho^{Mod}_{\Sigma}(\langle A, L, \varnothing\rangle) = A$$

Next, we prove that $\rho : \mathcal{FOPEQ} \to \mathcal{EVT}$ meets the axiomatic requirements of an institution comorphism as specified in Definition 214.

Theorem 32. *The institution comorphism* ρ *is defined such that for any* $\Sigma \in |\mathbf{Sign}_{\mathcal{FOPEQ}}|$*, the translations* $\rho^{Sen}_{\Sigma} : \mathbf{Sen}_{\mathcal{FOPEQ}}(\Sigma) \to \mathbf{Sen}_{\mathcal{EVT}}(\rho^{Sign}(\Sigma))$ *and* $\rho^{Mod}_{\Sigma} : \mathbf{Mod}_{\mathcal{EVT}}(\rho^{Sign}(\Sigma)) \to \mathbf{Mod}_{\mathcal{FOPEQ}}(\Sigma)$ *preserve the satisfaction relation. That is, for any* $\psi \in \mathbf{Sen}_{\mathcal{FOPEQ}}(\Sigma)$ *and* $M' \in |\mathbf{Mod}_{\mathcal{EVT}}(\rho^{Sign}(\Sigma))|$

$$\rho^{Mod}_{\Sigma}(M') \models_{\mathcal{FOPEQ}_{\Sigma}} \psi \ \Leftrightarrow \ M' \models_{\mathcal{EVT}_{\rho^{Sign}(\Sigma)}} \rho^{Sen}_{\Sigma}(\psi) \tag{3.1}$$

Proof. By Definition 38, $M' = \langle A, L, \varnothing\rangle$, $\rho^{Mod}_{\Sigma}(M') = A$ and $\rho^{Sen}_{\Sigma}(\psi) = \langle inv, \psi\rangle$. Therefore, Equation 3.1 becomes

$$A \models_{\mathcal{FOPEQ}_{\Sigma}} \psi \ \Leftrightarrow \ M' \models_{\mathcal{EVT}_{\rho^{Sign}(\Sigma)}} \langle inv, \psi\rangle$$

Then, by the definition of the satisfaction relation in $\mathcal{EVT}$ (Definition 37)

$$A \models_{\mathcal{FOPEQ}_{\Sigma}} \psi \ \Leftrightarrow \ A^{(s\prime)} \models_{\mathcal{FOPEQ}_{(\rho^{Sign}(\Sigma))^{(V\prime)}_{\mathcal{FOPEQ}}}} \psi$$

We deduce that $\Sigma = (\rho^{Sign}(\Sigma))^{V\prime}_{\mathcal{FOPEQ}}$, since there are no variable names in $V\prime$ and thus no new operator symbols are added to the signature. As there are no variable names in $V\prime$, $L = \{\{\}\}$, so we can conclude that $A^{(s\prime)} = A$. Thus the satisfaction condition holds. □

As described in Section 2.4.2, institution comorphisms allow us to translate specifications written over one institutions to specifications over another.

For a Σ-specification written over $\mathcal{FOPEQ}$ and institution comorphism $\rho : \mathcal{FOPEQ} \rightarrow \mathcal{EVT}$, we can use the specification-building operator

$$__ \ \texttt{with} \ \rho : Spec_{\mathcal{FOPEQ}}(\Sigma) \rightarrow Spec_{\mathcal{EVT}}(\rho^{Sign}(\Sigma))$$

to interpret this as a specification over $\mathcal{EVT}$ [100]. This results in a specification with just the `Init` event and no variables, containing $\mathcal{FOPEQ}$-sentences that hold in the initial state. This process is used to represent contexts, specifically their axioms, which are written over $\mathcal{FOPEQ}$ as sentences over $\mathcal{EVT}$.

In cases where a specification is enriched with new events, then the axioms and invariants should also apply to these new events. Our use of $\mathcal{EVT}$-sentences that define invariants mediates this as they are applied to all events in the specification when evaluating the satisfaction condition. In Section 2.4.1 we outlined that, in order for an institution to correctly utilise the specification-building operators, we must prove properties with regard to pushouts (Definition 211) and amalgamation (Definition 212). We prove that $\mathcal{EVT}$ meets these requirements in Section 3.7.

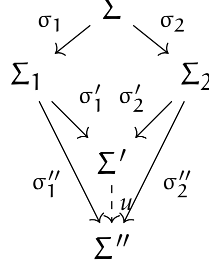
3.7 PUSHOUTS AND AMALGAMATION

We ensure that the institution $\mathcal{EVT}$ has good modularity properties by proving that $\mathcal{EVT}$ admits the amalgamation property: all pushouts in $\mathbf{Sign}_{\mathcal{EVT}}$ exist and every pushout diagram in $\mathbf{Sign}_{\mathcal{EVT}}$ admits *weak* model amalgamation [100].

Proposition 31. *Pushouts exist in* $\mathbf{Sign}_{\mathcal{EVT}}$.

Proof. Given two $\mathcal{EVT}$-signature morphisms $\sigma_1 : \Sigma \rightarrow \Sigma_1$ and $\sigma_2 : \Sigma \rightarrow \Sigma_2$ a pushout is a triple $(\Sigma', \sigma_1', \sigma_2')$ that satisfies the universal property: for all triples $(\Sigma'', \sigma_1'', \sigma_2'')$ there exists a unique morphism $u : \Sigma' \rightarrow \Sigma''$ such that the diagram on the left below commutes. Our pushout construction follows $\mathcal{FOPEQ}$ for the elements that $\mathcal{FOPEQ}$ has in common

with $\mathcal{EVT}$. In $\mathbf{Sign}_{\mathcal{EVT}}$ the additional elements are E and V as presented below.



We base our constructions of the pushout in E and V on the pushout in **Set** that was defined in Section 2.4.1.

- *Set of ⟨event name, status⟩ pairs E:* The set of all event names in the pushout is the pushout in **Set** on event names only. Then, the status of an event in the pushout is the supremum of all event statuses that are mapped to it, according to the ordering given in Definition 32. Since $\mathcal{EVT}$-signature morphisms map $\langle \texttt{Init,ordinary} \rangle$ to $\langle \texttt{Init,ordinary} \rangle$ the pushout does likewise. The universality property for E follows from that of **Set**. Thus the pushout in E is given by the formula $E_1 \dot{\cup} E_2 / \sim$ where $\sim$ is the least equivalence relation such that

$$\sigma_1' \circ \sigma_1(\langle e, status \rangle) \sim \sigma_2' \circ \sigma_2(\langle e, status \rangle)$$

  for $\langle e, status \rangle \in \Sigma.E$.

- *Set of sort-indexed variable names V:* The set of sort-indexed variable names in the pushout is the pushout in $\mathcal{FOPEQ}$ for the sort components and the pushout in **Set** for the variable names. This is a similar construction to the pushout for operation names in $\mathcal{FOPEQ}$ as these also have to follow the sort pushout. Thus, the universality property for V follows from that of **Set** and the $\mathcal{FOPEQ}$ pushout

for sorts.Thus the pushout in V is given by the formula $V_1 \dot\cup V_2 / \sim$ where $\sim$ is the least equivalence relation such that

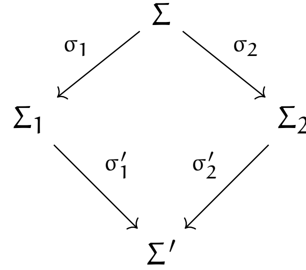
$$\sigma_1' \circ \sigma_1(v : s) \sim \sigma_2' \circ \sigma_2(v : s)$$

for $(v : s) \in \Sigma.V$.

□

In Definition 212 we defined what is meant by the amalgamation property. In Definition 39 we present the definition of amalgamation as outlined by Sanella and Tarlecki and use it to structure the proof of amalgamation in $\mathcal{EVT}$ [100]. Both definitions are equivalent but the latter allows us to format our proof in a more intuitive way.

Definition 39 (Amalgamation). Let $\mathbf{INS} = \langle \mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \langle \models_\Sigma \rangle_{\Sigma \in |\mathbf{Sign}|} \rangle$ be an institution. The following diagram in **Sign**



admits amalgamation if

- for any two models $M_1 \in |\mathbf{Mod}(\Sigma_1)|$ and $M_2 \in |\mathbf{Mod}(\Sigma_2)|$, there exists a unique model $M' \in |\mathbf{Mod}(\Sigma')|$ (amalgamation of M_1 and M_2) such that $M'|_{\sigma_1'} = M_1$ and $M'|_{\sigma_2'} = M_2$.
- for any two model morphisms $f_1 : M_{11} \to M_{12}$ in $\mathbf{Mod}(\Sigma_1)$ and $f_2 : M_{21} \to M_{22}$ in $\mathbf{Mod}(\Sigma_2)$ such that $f_1|_{\sigma_1} = f_2|_{\sigma_2}$, there exists a unique model morphism $f' : M_1' \to M_2'$ in $\mathbf{Mod}(\Sigma')$ (amalgamation of f_1 and f_2) such that $f'|_{\sigma_1'} = f_1$ and $f'|_{\sigma_2'} = f_2$.

The institution **INS** has the *amalgamation* property if all pushouts exist in **Sign** and every pushout diagram in **Sign** admits amalgamation [100].

Proposition 32. *Every pushout diagram in* $\textbf{Sign}_{\mathcal{EVT}}$ *admits weak model amalgamation.*

We decompose this proposition into two further sub-propositions:

Sub-Proposition 321. *For* $M_1 \in |\textbf{Mod}(\Sigma_1)|$ *and* $M_2 \in |\textbf{Mod}(\Sigma_2)|$ *such that* $M_1|_{\sigma_1} = M_2|_{\sigma_2}$*, there exists an* $\mathcal{EVT}$*-model (the amalgamation of* M_1 *and* M_2*)* $M' \in |\textbf{Mod}(\Sigma')|$ *such that* $M'|_{\sigma'_1} = M_1$ *and* $M'|_{\sigma'_2} = M_2$*.*

Proof. Given the $\mathcal{EVT}$-models $M \in |\textbf{Mod}(\Sigma)|, M_1 \in |\textbf{Mod}(\Sigma_1)|, M2 \in |\textbf{Mod}(\Sigma_2)|$ and $\mathcal{EVT}$-signature morphisms $\sigma_1 : \Sigma \to \Sigma_1, \sigma_2 : \Sigma :\to \Sigma_2$ in the commutative diagram below.

$$
\begin{array}{ccccc}
 & & M' = \langle A', L', R' \rangle & & \\
 & \swarrow^{Mod(\sigma'_1)} & & \searrow^{Mod(\sigma'_2)} & \\
M_1 = \langle A_1, L_1, R_1 \rangle & & & & M_2 = \langle A_2, L_2, R_2 \rangle \\
 & \searrow_{Mod(\sigma_1)} & & \swarrow_{Mod(\sigma_2)} & \\
 & & M = \langle A, L, R \rangle & &
\end{array}
$$

We compute the model amalgamation (following the corresponding pushout diagram in **Sign** as illustrated in Proposition 31) $M' = M_1 \otimes M_2$ which is of the form $\langle A', L', R' \rangle$ where $A' = A_1 \otimes A_2$ is the $\mathcal{FOPEQ}$-model amalgamation of A_1 and A_2. We construct the initialising set $L' = L_1 \otimes L_2$ by amalgamating the initialising sets L_1 and L_2 to get the set of all possible combinations of variable mappings, while respecting the amalgamations induced on variable names via the pushout $\Sigma.V'$.

For example, suppose that the sort-indexed variable x is in $\Sigma.V$. Then we can apply the $\mathcal{EVT}$-signature morphisms $\sigma_1 : \Sigma \to \Sigma_1$ and $\sigma_2 : \Sigma \to \Sigma_2$ to get $\sigma_1(x) = x_1 \in \Sigma_1.V$ and $\sigma_2(x) = x_2 \in \Sigma_2.V$. From Proposition 31, we know that the pushout $(\Sigma_1.V \cup \Sigma_2.V)/_\sim$ contains the variable name x' such that $\sigma'_1(x_1) = \sigma'_2(x_2) = x' \in \Sigma'.V$. We follow this pushout construction in order to construct the corresponding model amalgamation. A model $M \in |\textbf{Mod}(\Sigma)|$ contains an initialising set L which, in turn, contains maplets of the form $x \mapsto a$ where a is a sort-appropriate value for

the variable x. In light of the $\mathcal{EVT}$-signature morphisms outlined above, we know that $M_1 \in |\mathbf{Mod}(\Sigma_1)|$ contains maplets of the form $x_1 \mapsto a_1$ and $M_2 \in |\mathbf{Mod}(\Sigma_2)|$ contains maplets of the form $x_2 \mapsto a_2$ where a_1 and a_2 are sort-appropriate values for x_1 and x_2 respectively. Then the amalgamation $M' \in |\mathbf{Mod}(\Sigma)|$ has the initialising set $L' = L_1 \otimes L_2$ which contains all maplets of the form $x' \mapsto a'$ where a' is a sort-appropriate value for x' drawn from A' (the $\mathcal{FOPEQ}$-amalgamation of A_1 and A_2). Note that initialising sets contain sets of variable-to-value mappings for all of the variables in the V component of their signatures, as such, the amalgamation contains the above maplets in all possible combinations with those corresponding to any other variables that may be in the signature.

We construct the relation $R' = R_1 \otimes R_2$, which is the amalgamation of R_1 and R_2, in a similar manner. Specifically, starting from any $R.e = \{s_1, ..., s_m\} \in R$ where $s_1, ..., s_m$ are states of the form

$$\{x_1 \mapsto a_1, ..., x_{n'} \mapsto a_{n'}\}$$

where $x_1, \ldots, x_{n'}$ are the variable names (and their primed versions drawn from $\Sigma.V$). We construct the corresponding relation $R'.\sigma'(e)$ in R' so that the diagram in Figure 3.4 commutes.

In Figure 3.4, $h' = (h_1 + h_2)$ is the corresponding function over the carrier-sets in M' obtained from $\mathcal{FOPEQ}$, and $\sigma' = (\sigma'_1 \circ \sigma_1) + (\sigma'_2 \circ \sigma_2)$ is the mapping for variable and event names obtained from the corresponding construction in $\mathbf{Sign}_{\mathcal{EVT}}$. $\square$

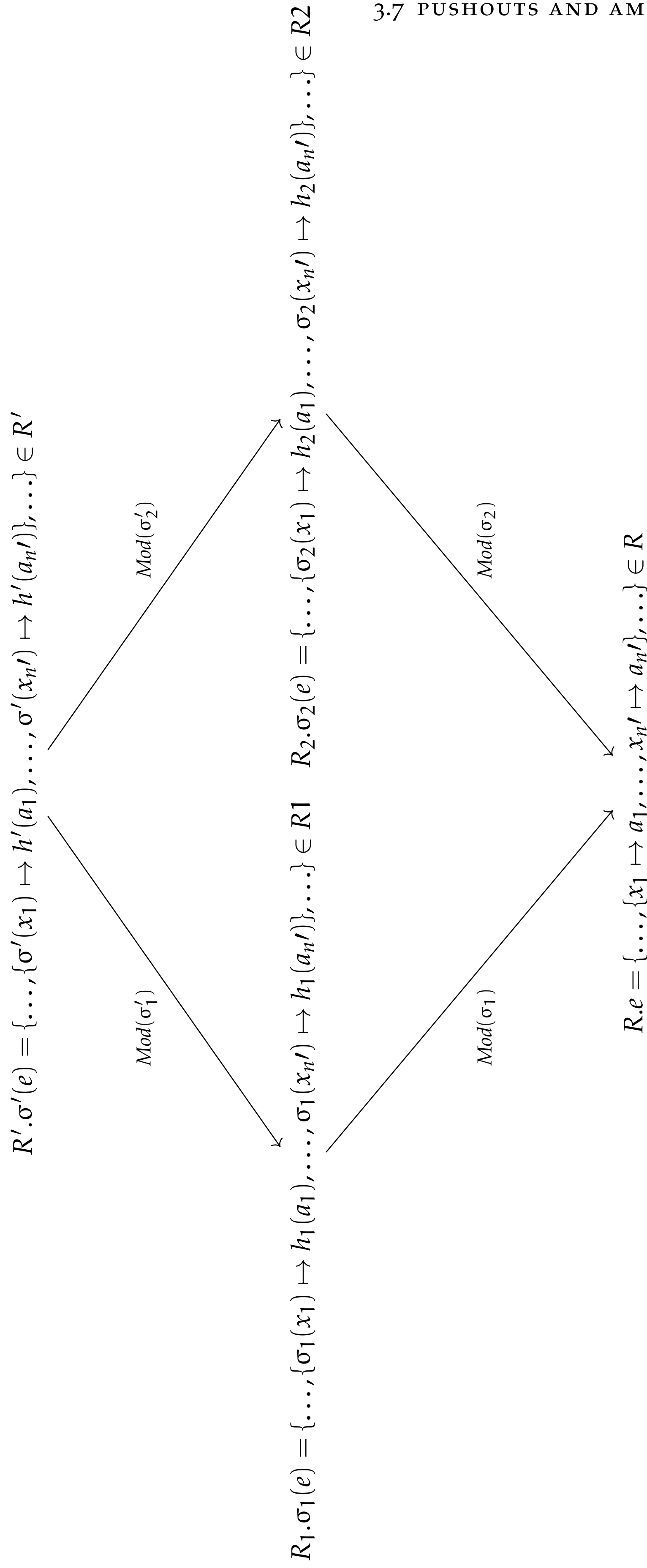


Figure 3.4: The construction of $R' = R_1 \otimes R_2$, the amalgamation of R_1 and R_2.

Sub-Proposition 322. *For any two $\mathcal{EVT}$-model morphisms $f_1 : M_{11} \to M_{12}$ in $\boldsymbol{Mod}(\Sigma_1)$ and $f_2 : M_{21} \to M_{22}$ in $\boldsymbol{Mod}(\Sigma_2)$ such that $f_1|_{\sigma_1} = f_2|_{\sigma_2}$, there exists an $\mathcal{EVT}$-model morphism (the amalgamation of f_1 and f_2) called $f' : M_1' \to M_2'$ in $\boldsymbol{Mod}(\Sigma')$, such that $f'|_{\sigma_1'} = f_1$ and $f'|_{\sigma_2'} = f_2$.*

Proof. Given the $\mathcal{EVT}$-model morphisms f_1 and f_2 and their common reduct f_0, we construct f' so that the following diagram commutes:

$$
\begin{array}{ccccc}
 & & f' : M_1' \to M_2' & & \\
 & \swarrow^{Mod(\sigma_1')} & & \searrow^{Mod(\sigma_2')} & \\
f_1 : M_{11} \to M_{12} & & & & f_2 : M_{21} \to M_{22} \\
 & \searrow_{Mod(\sigma_1)} & & \swarrow_{Mod(\sigma_2)} & \\
 & & f_0 : M_{01} \to M_{02} & &
\end{array}
$$

Since each $\mathcal{EVT}$-model has a $\mathcal{FOPEQ}$ model as its first component, each of the $\mathcal{EVT}$-model morphisms f_0, f_1, f_2 and f' must have an underlying $\mathcal{FOPEQ}$-model morphism, which we denote f_0^-, f_1^-, f_2^- and f'^- respectively. To build the amalgamation for $\mathcal{EVT}$-models we must show how to extend these to cover the data states of the $\mathcal{EVT}$-models. This $\mathcal{EVT}$-model morphism follows the underlying $\mathcal{FOPEQ}$-model morphism on sort carrier sets for the values in the data states.

Given $R.e \in R$, suppose we start with any f_0-maplet of the form

$$
\begin{aligned}
&\{\ldots, \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}, \ldots\} \\
&\quad \mapsto \{\ldots, \{x_1 \mapsto f_0^-(a_1), \ldots, x_n \mapsto f_0^-(a_n)\}, \ldots\}_e \; \in \; f_0
\end{aligned}
$$

where f_0^- is the underlying map on data types from the $\mathcal{FOPEQ}$-model morphism.

Then the original two functions in f_1 and f_2 must have maplets of the form

$$
\begin{aligned}
&\{\ldots, \{\sigma_1(x_1) \mapsto h_1(a_1), \ldots, \sigma_1(x_{n\prime}) \mapsto h_1(a_{n\prime})\}, \ldots\} \\
&\quad \mapsto \{\ldots, \{\sigma_1(x_1) \mapsto f_1^-(h_1(a_1)), \ldots, \sigma_1(x_{n\prime}) \mapsto f_1^-(h_1(a_{n\prime}))\}, \ldots\} \; \in \; f_1
\end{aligned}
$$

and

$$\{\ldots,\{\sigma_2(x_1) \mapsto h_2(a_1),\ldots,\sigma_2(x_{n'}) \mapsto h_2(a_{n'})\},\ldots\}$$
$$\mapsto \{\ldots,\{\sigma_2(x_1) \mapsto f_2^-(h_2(a_1)),\ldots,\sigma_2(x_{n'}) \mapsto f_2^-(h_2(a_{n'}))\},\ldots\} \in f_2$$

where f_1^- and f_2^- are again the data type maps from the underlying $\mathcal{FOPEQ}$-model morphism, and h_1 and h_2 are obtained from $Mod(\sigma_1)$ and $Mod(\sigma_2)$.

We then can construct the elements of the $\mathcal{EVT}$-model morphism f', which is the amalgamation of f_1 and f_2, as f'-maplets of the form:

$$\{\ldots,\{\sigma'(x_1) \mapsto h'(a_1),\ldots,\sigma'(x_{n'}) \mapsto h'(a_{n'})\},\ldots\}$$
$$\mapsto \{\ldots,\{\sigma'(x_1) \mapsto f'^-(h'(a_1)),\ldots,\sigma'(x_{n'}) \mapsto f'^-(h'(a_{n'}))\},\ldots\} \in f'$$

As before, $h' = (h_1 + h_2)$ is the corresponding function over the carrier-sets in M' obtained from $\mathcal{FOPEQ}$, and $\sigma' = (\sigma_1' \circ \sigma_1) + (\sigma_2' \circ \sigma_2)$ is the mapping for variable and event names obtained from the corresponding construction in **Sign**. Here $f'^- = f_1^- + f_2^-$ is the model morphism amalgamation from the corresponding diagram for model morphisms in $\mathcal{FOPEQ}$, which ensures that the data states are mapped to corresponding states in the model M_2'.

□

So far, we have defined our institution for Event-B, $\mathcal{EVT}$, outlined the comorphism from $\mathcal{FOPEQ}$ to $\mathcal{EVT}$, and shown that $\mathcal{EVT}$ meets the requirements needed for correct use of the specification-building operators. Next, we discuss the pragmatics of specification-building in $\mathcal{EVT}$ (Section 3.8), and illustrate, by example, how specifications can be written and modularised in $\mathcal{EVT}$ (Section 3.9).

3.8 PRAGMATICS OF SPECIFICATION BUILDING IN $\mathcal{EVT}$

We represent an Event-B specification, that is composed of machines and contexts, as a *presentation* (Definition 29) over $\mathcal{EVT}$. Technically, $\mathcal{EVT}$ allows for loose specifications which are not possible in Event-B, however, we do not exploit this in our examples that follow.

Recall that, for any signature Σ, a Σ-presentation is a set of Σ-sentences. A model of a Σ-presentation is a Σ-model that satisfies all of the sentences in the presentation [52]. Thus, for a presentation in $\mathcal{EVT}$, model components corresponding to an event must satisfy all of the sentences specifying that event. This incorporates the standard semantics of the *extends* operator for events in Event-B where the extending event implicitly contains all the parameters, guards and actions of the extended event but can include additional parameters, guards and actions [5].

An interesting aspect is that if a variable is not assigned a value, within an action, then a model for the event may associate a new value with this variable. Some languages deal with this using a *frame condition*, asserting implicitly that values for unmodified variables do not change. In Event-B such a condition would cause complications when combining presentations, since variables unreferenced in one event will be constrained not to change, and this may contradict an action for them in the other event. As far as we can tell, the informal semantics for the Event-B language does not require a frame condition, and we have not included one in our definition.

In the next section, we show how the traffic light Event-B specification in Figure 2.1 can be represented in $\mathcal{EVT}$.

3.9 WRITING SPECIFICATIONS IN THE $\mathcal{EVT}$ INSTITUTION

We now introduce the HETS-style notation that we will use to write specifications over the $\mathcal{EVT}$ institution. Here, we show how the traffic light example introduced in Figure 2.1 (Chapter 2) can be written and modularised in this setting. Our definition of $\mathcal{EVT}$ allows the restructuring of Event-B specifications using the standard specification-building operators for institutions [100]. Thus $\mathcal{EVT}$ provides a means for writing down and splitting up the components of an Event-B system, facilitating increased modularity for Event-B specifications. Figure 3.5 contains heterogeneous structured specifications corresponding to the Event-B machine `mac1` defined in Figure 2.1 (lines 1–21). Since HETS is our target platform, where each institution is represented as a logic, we use its notation and implementation of the logic for $\mathcal{CASL}$ to represent the $\mathcal{FOPEQ}$ components of our specifications.

LINES 1–6: `TwoBools` can be presented as a pure $\mathcal{CASL}$ specification, declaring two boolean variables constrained to have different values.

LINES 7–17: `LightAbstract` is a specification in the $\mathcal{EVT}$ logic for a single traffic light that extends (using keyword `then`) `TwoBools`. On line 9, we can see that `TwoBools` is first translated via the comorphism ρ into a specification over $\mathcal{EVT}$. It contains the events `set_go` and `set_stop`, with the constraint that a light can only be set to `go` if its opposite light is not set to `go`. We use `thenAct` in place of the `then` Event-B keyword to distinguish from the `then` specification-building operator.

LINES 18–32: The specification `mac1` combines (using keyword `and`) two versions of `LightAbstract`, each `with` a different signature morphism (σ_1 and σ_2) mapping the specification variables and event names to those in the Event-B machine. The `where` notation used on lines 22–32 is a convenient presentation of the signature morphisms, it is not part of the syntax of the specification language that we use in HETS.

```
1  logic CASL
2  spec  TwoBools =
3    Bool
4    then
5      ops i_go, u_go : Bool
6      . ¬ (i_go = true ∧ u_go = true)

7  logic EVT
8  spec  LightAbstract =
9    TwoBools with ρ
10   then
11     Initialisation ordinary
12       thenAct  act1 : i_go := false
13     Event set_go ≙ ordinary
14       when  grd1: u_go = false
15       thenAct  act1: i_go := true
16     Event set_stop ≙ ordinary
17       thenAct  act1: i_go := false

18 logic EVT
19 spec  mac1 =
20   (LightAbstract with σ1)
21     and (LightAbstract with σ2)
22   where
23     σ1 = {i_go ↦ cars_go, u_go ↦ peds_go,
24          ⟨set_go, ordinary⟩
25            ↦ ⟨set_cars_go, ordinary⟩,
26          ⟨set_stop, ordinary⟩
27            ↦ ⟨set_cars_stop, ordinary⟩}
28     σ2 = {i_go ↦ peds_go, u_go ↦ cars_go,
29          ⟨set_go, ordinary⟩
30            ↦ ⟨set_peds_go, ordinary⟩,
31          ⟨set_stop, ordinary⟩
32            ↦ ⟨set_peds_stop, ordinary⟩}
```

Figure 3.5: A modular institution-based presentation corresponding to the abstract machine mac1 in Fig 2.1.

This results in a presentation over the institution $\mathcal{EVT}$ for mac1 by flattening out the structuring. Notice that the specification for each individual light had to be explicitly written down twice in the Event-B machine in Figure 2.1 (lines 11–15 and lines 16–20). In our modular institution-based presentation we only need one light specification and simply supply the required variable and event mappings. In this way, $\mathcal{EVT}$ provides a more flexible degree of modularity than is currently present in Event-B.

Figure 3.6 contains a presentation over $\mathcal{EVT}$ corresponding to the main elements of the Event-B specification mac2 presented in Figure 2.1 (lines 22–59). Here, we present three $\mathcal{CASL}$ specifications and three $\mathcal{EVT}$ specifications.

LINES 1–10: We specify the Colours data type with a standard $\mathcal{CASL}$ specification, as can be seen in the context specification on lines 22–27 of Figure 2.1. The specification TwoColours describes two variables of type Colours constrained to be not both green at the same time. This corresponds to the gluing invariants on lines 33 and 35 of Figure 2.1. The specification modularisation constructs used in Figure 3.6, allow these properties to be handled distinctly and in a manner that

```
1  logic CASL
2  spec  Colours =
3    then
4      sorts
5      free type Colours  ::=  red|green|
                                 orange

6  spec  TwoColours =
7      Colours
8      then
9        ops icol, ucol :  Colours
10       . ¬ (icol = green  ∧  ucol = green)

11 spec  BoolButton =
12   Bool
13   then
14     ops button :  Bool

15 logic EVT
16 spec  LightRefined =
17   TwoColours with ρ
18   then
19     Initialisation ordinary
20       thenAct  act1: icol  :=  red
21     Event set_green ≙ ordinary
22       when  grd1: ucol  =  red
23       thenAct  act1: icol  :=  green
24     Event set_red ≙ ordinary
25       thenAct  act1: icol  :=  red

26 logic EVT
27 spec  ButtonSpec =
28   BoolButton with ρ
29   then
30     Event gobutton ≙ ordinary
31       when  grd1: button  =  true
32       thenAct  act1: button  :=  false
33     Event pushbutton ≙ ordinary
34       thenAct  act1: button  :=  true

35 spec  mac2 =
36   (LightRefined  with σ3)
37 and (LightRefined and
38        (ButtonSpec with σ5) with σ4)

39 where
40   σ3 = {i_col ↦ cars_colour,  u_col ↦ peds_colour,
41         ⟨set_green, ordinary⟩
42           ↦  ⟨set_cars_green, ordinary⟩,
43         ⟨set_red, ordinary⟩
44           ↦  ⟨set_cars_red, ordinary⟩}
45   σ4 = {i_col ↦ peds_colour,  u_col ↦ cars_colour,
46         ⟨set_green, ordinary⟩
47           ↦  ⟨set_peds_green, ordinary⟩,
48         ⟨set_red, ordinary⟩
49           ↦  ⟨set_peds_red, ordinary⟩}
50   σ5 = {⟨gobutton, ordinary⟩
51           ↦  ⟨set_green, ordinary⟩}
```

Figure 3.6: A modular institution-based presentation corresponding to the refined machine `mac2` specified in Figure 2.1 (lines 28–59).

facilitates comparison with the `TwoBools` specification on lines 1–6 of Figure 3.5.

LINES 15–25: A specification for a single light is provided in `LightRefined` which uses `TwoColours` to describe the colour of the lights. As was the case with `LightAbstract` in Figure 3.5, the specification makes clear how a single light operates. An added benefit here is that a direct comparison with the abstract specification can be done on a per-light basis.

LINES 11–14, 26–34: The specifications `BoolButton` and `ButtonSpec` account for the part of the `mac2` specification that requires a button. These details were woven through the code in Figure 2.1 (lines 30, 36, 44, 46, 57, 58) but the specification-building operators allow us to modularise the specification and group these related definitions together, clarifying how the button actually operates.

LINES 35–51: Finally, to bring this all together we combine a copy of `LightRefined` with a specification corresponding to the sum (`and`) of `LightRefined` and `ButtonSpec` `with` appropriate signature morphisms. This second specification combines the event `gobutton` in `ButtonSpec` with the event `set_green` in `LightRefined` thus accounting for `set_peds_green` in Figure 2.1. One small issue involves making sure that the name replacements are done correctly, and in the correct order, hence the bracketing on lines 37–38 is important.

The combination of these specifications involves merging two events with different names: `gobutton` from `ButtonSpec` with the event `set_green` from `LightRefined`. To ensure that these differently-named events are combined into an event of the same name we use the signature morphism σ_5 to give `gobutton` the same name as `set_green` before combining them. Ensuring that the events have the same name allows the `and` operator to combine both events' guards and actions and the morphism σ_4 to name the resulting event `set_peds_green`. The resulting specification also contains the event `pushbutton`. The labels given to guards/actions are syntactic sugar to make the specification aesthetically resemble the usual Event-B notation for guards/actions.

3.9.1 REPRESENTING REFINEMENT EXPLICITLY

As outlined in Chapters 1 and 2, refinement is a central aspect of the Event-B methodology, therefore any formalisation of Event-B must be capable of capturing refinement. In Section 2.3.4, we described how the institutional framework accounts for Event-B refinement. Figure 3.7 uses the refinement syntax available in HETS to specify each of the refinements in the specification of the concrete machine `mac2`:

LINES 2–4: define the data refinement of Bool into Colours, with an appropriate mapping for the values.

```
1  refinement REF : Bool to Colours =
2      Bool  ↦ Colours,
3      true  ↦ green,
4      false ↦ red
5      i_go ↦ icol,
6      u_go ↦ ucol,
7      ⟨set_peds_go, ordinary⟩
8          ↦ ⟨set_peds_green, ordinary⟩,
9      ⟨set_peds_stop, ordinary⟩
10         ↦ ⟨set_peds_red, ordinary⟩,
11     ⟨set_cars_go, ordinary⟩
12         ↦ ⟨set_cars_green, ordinary⟩,
13     ⟨set_cars_stop, ordinary⟩
14         ↦ ⟨set_cars_red, ordinary⟩
15 end
```

Figure 3.7: Defining the refinement relationships between the concrete and abstract presentations.

LINES 5–6: define the refinement of the two boolean variables into their corresponding variables of type `Colour`. In combination with lines 2–4, this corresponds to the gluing invariants on lines 33 and 35 of Figure 2.1.

LINES 7–14: define the refinement relation between the four events: this corresponds to the `refines` statements on lines 42, 48, 51 and 55 of Figure 2.1.

Thus, the Event-B specification of a traffic-lights system, that was introduced in Chapter 2 can be captured and modularised using $\mathcal{EVT}$.

3.10 SUMMARY

In this chapter, we have presented our definition of an institution for Event-B, $\mathcal{EVT}$ and the associated set of proofs to show that it preserves the axiomatic requirements of an institutional definition. This institution is an updated version of our original definition of an institution for Event-B that includes a new kind of sentence for invariants and their associated satisfaction relation [40, 41]. We have also introduced the HETS-style notation that we will use for writing (modular) $\mathcal{EVT}$-specifications. In the next chapter we formalise a translational semantics for Event-B by translating Event-B specifications to $\mathcal{EVT}$-specifications.

4

FORMALISING A TRANSLATIONAL SEMANTICS FOR EVENT-B

In this chapter we formalise a translational semantics for Event-B, using the institution that we have defined for Event-B, $\mathcal{EVT}$ in Chapter 3. Our approach involves outlining a three layer model of the Event-B language by splitting it into its mathematical, infrastructure and superstructure sub-languages. Then, we provide a translational semantics for each of these sub-languages.

4.1 INTRODUCTION

In Section 2.1.2 we remarked that, although there is no fixed semantics for Event-B models, the semantics is generally provided implicitly by the proof obligations that Rodin generates for the model [59]. This can be advantageous in that it allows the use of similar proof obligations across multiple modelling domains. We propose a translational semantics of Event-B from specifications written in Event-B to specifications written in the institutional language of $\mathcal{EVT}$. This does not inhibit the freedom of being able to use Event-B as a calculus in diverse modelling domains but rather amplifies it.

It has been shown that a shallow embedding of the Event-B logic in Isabelle/HOL provides a sound semantics for the base logic of Event-B but not for the full modelling language [102]. Moreover, Event-B is a special form of Back's Action systems which has been formally defined using Dijkstra's weakest precondition predicate transformers [8]. This

accounts for the refinement calculus used but not for a full semantic definition of the Event-B modelling language itself.

4.2 SYNTAX OF EVENT-B

Our objective is to define a translational semantics for Event-B by representing Event-B specifications as specifications over $\mathcal{EVT}$, our institution for Event-B [40, 41]. As described earlier, there are two basic languages in Event-B, these are the Event-B mathematical language (propositional/predicate logic, set theory and arithmetic) and the Event-B modelling language [3]. We decompose the Event-B modelling language into two further languages which (inspired by the UML specification) we call the *infrastructure language* and the *superstructure language*. In order to translate Event-B into the institutional framework we divide the constructs of the Event-B language into three layers, each layer corresponding to one of its three constituent languages, as illustrated in Figure 4.1.

- At the base of Figure 4.1 is the Event-B mathematical language. The institution for first-order predicate logic with equality, $\mathcal{FOPEQ}$, is embedded via an institution comorphism into the institution for Event-B, $\mathcal{EVT}$ (Theorem 32). The semantics that we define translates the constructs of this mathematical language into corresponding constructs over $\mathcal{FOPEQ}$.

- At the next level is the Event-B infrastructure, which consists of those language elements used to define variables, invariants, variants and events. These are translated into sentences over $\mathcal{EVT}$ (Definition 33).

- At the topmost level is the Event-B superstructure which deals with the definition of Event-B machines and contexts, as well as

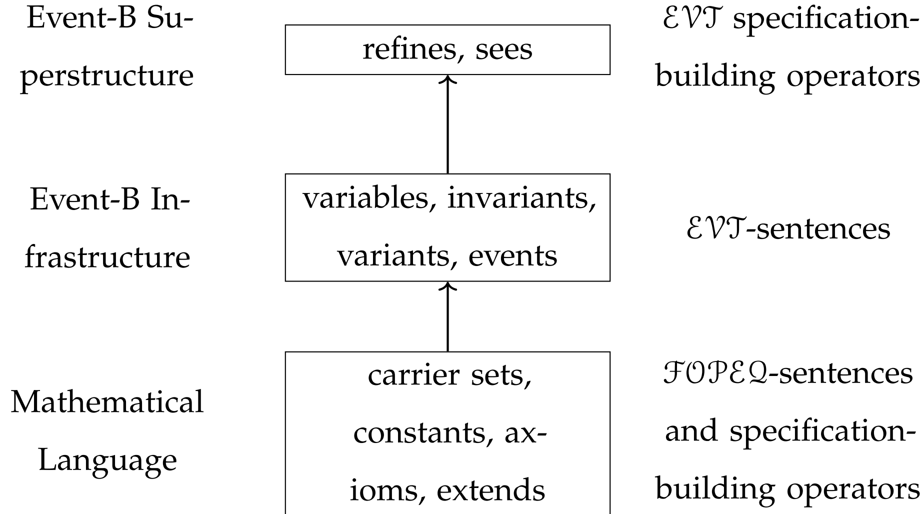


Figure 4.1: We split the Event-B syntax into three components: superstructure, infrastructure and a mathematical language

their relationships (`refines`, `sees`, `extends`). These are translated into presentations over $\mathcal{EVT}$.

The abstract syntax for Event-B is described briefly in [3] and we provide a more detailed version in Figure 4.2. A *Specification* consists of any number of *Machine* and *Context* definitions. The nonterminals *predicate* and *expression* are not defined in Figure 4.2. These are part of the Event-B mathematical language, and in our translation these syntactic elements will be supplied by $\mathcal{FOPEQ}$, the institution for first-order predicate logic with equality as described in Section 2.3.3, with predicates corresponding to $\mathcal{FOPEQ}$-formulae and expressions corresponding to $\mathcal{FOPEQ}$-terms.

Both machines and contexts allow the user to specify *theorems* which are used to generate proof obligations. Since these must be consequences of the specification and do not add any constraints, we omit them from further discussion here. We order things in a slightly different manner to the standard in Event-B in that we use *MachineBody*,

EventBody and *ContextBody* to refer to the non-superstructure elements of a machine, event or context.

Based on the syntax defined in Figure 4.2, we define the semantics of each of the Event-B infrastructure sentences by describing a mechanism to translate them into $\mathcal{EVT}$-sentences. In order to carry out such a translation we first extract the corresponding $\mathcal{EVT}$ signature $\Sigma = \langle S, \Omega, \Pi, E, V \rangle$ from any given Event-B specification. Contexts can be represented entirely by the underlying mathematical language and thus translated into specifications over $\mathcal{FOPEQ}$. In Section 4.3, we define the interface in Figure 4.3 in order to facilitate the use of some $\mathcal{FOPEQ}$ operations and semantic functions.

We provide semantic functions for extracting the signature of an Event-B specification in Section 4.4. Sections 4.5 and 4.6 define the semantics of the Event-B superstructure and infrastructure languages respectively.

4.3 A $\mathcal{FOPEQ}$ INTERFACE

The Event-B formalism is parametrised by an underlying mathematical language and $\mathcal{EVT}$, our institution for Event-B is parametrised by $\mathcal{FOPEQ}$, the institution for first-order predicate logic with equality. In Figure 4.3, we define a $\mathcal{FOPEQ}$ interface in order to facilitate the use of its operations and semantic functions within our semantic definition of Event-B using $\mathcal{EVT}$. The description of these operations is outlined in Figure 4.3 and not specified further.

The semantic function $\mathbb{P}_\Sigma$ described in Figure 4.3 takes a labelled predicate and outputs a $\mathcal{FOPEQ}$-sentence (Σ-*formula*). The semantic function $\mathbb{T}_\Sigma$ takes an expression and returns a Σ-*term*. These functions are used later to translate Event-B predicates and expressions into Σ-*formulae* and Σ-*terms* respectively.

```
Specification ::= (Machine | Context)+
Machine       ::= machine identifier
                  [refines identifier]
                  [sees identifier+]
                  MachineBody
                  end
MachineBody   ::= variables identifier+
                  invariants LabelledPred*
                  [theorems LabelledPred+]
                  [variant expression]
                  events InitEvent Event*
LabelledPred  ::= label : predicate
InitEvent     ::= event Initialisation
                  status ordinary
                  [then LabelledPred]
                  end
Event         ::= event identifier
                  status Stat
                  [refines identifier+]
                  EventBody
                  end
EventBody     ::= [any identifier+]
                  [where LabelledPred]
                  [with LabelledPred]
                  [then LabelledPred]
Stat          ::= ordinary | convergent
                  | anticipated

Context       ::= context identifier
                  [extends identifier+]
                  ContextBody
                  end
ContextBody   ::= [sets identifier+]
                  [constants identifier+]
                  [axioms LabelledPred+]
                  [theorems LabelledPred+]
identifier    ::= String
label         ::= String
```

Figure 4.2: The Event-B syntax is parametrised by first-order logic as indicated by our use of the nonterminals *predicate* and *expression*. These will be mapped to $\mathcal{FOPEQ}$-formulae and terms respectively in our translational semantics.

The purpose of the semantic function $\mathbb{M}$ is to take two lists of identifiers and a list of labelled predicates and, use these to form the $\mathcal{FOPEQ}$ signature $\langle S, \Omega, \Pi \rangle$. The reason for this is that when extracting a signature from a context (described in Figure 4.9) carrier sets are interpreted as sorts and used to form S. The constants and axioms are used to form Ω and Π. We assume that $\mathbb{M}$ provides this translation.

For simplicity, we assume that it is possible to use Event-B identifiers in $\mathcal{FOPEQ}$ and $\mathcal{EVT}$. Also, when we reference a Σ-*formula* in Figure 4.3 we mean a possibly open $\mathcal{FOPEQ}$-formula over the signature given by Σ. We only return a closed $\mathcal{FOPEQ}$-formula when applying $\mathbb{P}_\Sigma$ to axiom sentences since they form closed predicates in Figure 4.9.

Of course, in order to from sentences in any institution, we must first extract the signature.

4.4 EXTRACTING THE SIGNATURE

We define an environment *Env* to map machine/context names to signatures, since, due to the superstructure components, machines/contexts can refer to other machines/contexts. In all further definitions we use ξ to denote an environment as defined by *Env*. We define the overloaded semantic function $\mathbb{D}$ in Figures 4.4 and 4.5 to extract the environment from a given specification (machines and contexts respectively).

We map $\mathbb{D}$ through the list of machines and contexts that make up an Event-B specification. $\mathbb{D}$ extracts the signature from machines and contexts. The function *def* (Figure 4.4) extracts the pair $\langle$event name, status$\rangle$ for each event in the machine and these pairs form the E component of the signature. The status is paired with each event name in order to correctly form variant sentences which will be discussed in Section 4.6. The function *ref* (Figure 4.4) forms the set of events that a particular concrete event refines. We use this function to remove the

$\mathcal{FOPEQ}$ **Operations**

- `F.and` : Σ-*formula** $\rightarrow$ Σ-*formula*
  This corresponds to the logical conjunction ($\bigwedge$) of a set of formulae in $\mathcal{FOPEQ}$.
- `F.lt` : Σ-*formula* $\times$ Σ-*formula* $\rightarrow$ Σ-*formula*
  This operation takes two formulae and returns a formula corresponding to arithmetic less than ($<$).
- `F.leq` : Σ-*formula* $\times$ Σ-*formula* $\rightarrow$ Σ-*formula*
  This operation takes two formulae and returns a formula corresponding to arithmetic less than or equal to ($\leqslant$).
- `F.exists` : *identifier** $\times$ Σ-*formula* $\rightarrow$ Σ-*formula*
  This operation takes a sequence of identifiers and a formula and returns a formula corresponding to the existential quantification of the identifiers over the input formula.
- `F.ι` : *identifier** $\rightarrow$ Σ-*formula* $\rightarrow$ Σ-*formula*
  This operation takes a list of identifiers and formula and returns the input formula with the names of all the free variables (as given by the list of identifiers) primed.

$\mathcal{FOPEQ}$ **Semantic Functions**

- $\mathbb{P}_\Sigma$: *LabelledPred* $\rightarrow$ Σ-*formula*
- $\mathbb{T}_\Sigma$: *expression* $\rightarrow$ Σ-*term*
- $\mathbb{M}$: *identifier** $\times$ *identifier** $\times$ *LabelledPred** $\rightarrow$ $|\mathbf{Sign}_{\mathcal{FOPEQ}}|$

Figure 4.3: The $\mathcal{FOPEQ}$ interface provides access to a range of operations and semantic functions which we assume to exist. These are used throughout our semantic definitions in Figures 4.4, 4.5, 4.7, 4.8, 4.9 and 4.10.

names of the refined abstract events, and the status that each is paired with, from the abstract machine's signature before we combine it with the concrete signature. We use the domain anti-restriction operator $\lhd$ to express this.

For an Event-B specification SP, we form an environment $\xi = \mathbb{D}[\![SP]\!]\xi_0$ *where* ξ_0 *is the empty environment.*

- $Env = Id \rightarrow |\,\mathbf{Sign}\,|$ # *An environment maps machine/context names to their signatures*

- $\mathbb{D} : Specification \rightarrow Env \rightarrow Env$

  $\mathbb{D}\ [\![\langle\,\rangle]\!]\ \xi = \xi$

  $\mathbb{D}\ [\![hd :: tl]\!]\ \xi = \mathbb{D}\ ([\![tl]\!])\ (\mathbb{D}\ [\![hd]\!]\ \xi)$

- $\mathbb{D} : Machine \rightarrow Env \rightarrow Env$ # *Extract and store the signature for a machine*

  $$\mathbb{D}\left[\!\!\left[\begin{array}{l}\texttt{machine } m\\ \texttt{refines } a\\ \texttt{sees } ctx_1, \ldots, ctx_n\\ mbody\\ \texttt{end}\end{array}\right]\!\!\right]\xi = \xi \cup \{[\![m]\!] \mapsto (\langle S, \Omega, \Pi, E, V\rangle \cup r(\xi[\![a]\!]))\}$$

  where

  $\langle S, \Omega, \Pi\rangle = \{(\xi\ [\![ctx_1]\!]) \cup \ldots \cup (\xi\ [\![ctx_n]\!])\}$ # *Include signatures from 'seen' contexts*

  $\langle E, V, RA\rangle = \mathbb{D}[\![mbody]\!]$ # *Collect names from machine body*

  $r : |\,\mathbf{Sign}\,| \rightarrow |\,\mathbf{Sign}\,|$ # *Include the signature for the abstract machine (less the refined events)*

  $r(\xi[\![a]\!]) = \texttt{let } \Sigma_a = \xi[\![a]\!] \texttt{ in } \langle \Sigma_a.S, \Sigma_a.\Omega, \Sigma_a.\Pi, RA \lhd \Sigma_a.E, \Sigma_a.V\rangle$

- $\mathbb{D} : MachineBody \rightarrow \langle E, V, \{identifier\}\rangle$ # *Extract signature elements from machine-body*

  $$\mathbb{D}\left[\!\!\left[\begin{array}{l}\texttt{variables } v_1, \ldots, v_n\\ \texttt{invariants } i_1, \ldots, i_n\\ \texttt{theorems } t_1, \ldots, t_n\\ \texttt{variant } n\\ \texttt{events } e_{init}\ e_1, \ldots, e_n\end{array}\right]\!\!\right] = \langle E, V, RA\rangle$$

  where

  $E = \{def[\![e_{init}]\!], def[\![e_1]\!], \ldots, def[\![e_n]\!]\}$ # *Names of events defined here*

  $V = \langle [\![v_1]\!], \ldots, [\![v_n]\!]\rangle$ # *Names of variables declared here*

  $RA = ref[\![e_{init}]\!] \cup ref[\![e_1]\!] \cup \ldots \cup ref[\![e_n]\!]$ # *Names of (abstract) events refined here*

- $def : Event \rightarrow identifier \times Stat$ # *Extract event name & status from an event definition*

  $def\ [\![\texttt{event } e, \texttt{status } s, \texttt{refines } e_1, \ldots, e_n, \cdots \texttt{end}]\!] = \langle [\![e]\!], s\rangle$

  $def\ [\![e_{init}]\!] = \langle \texttt{Initialisation}, \texttt{ordinary}\rangle$

- $ref : Event \rightarrow \{identifier\}$ # *Extract names of refined events from an event definition*

  $ref\ [\![\texttt{event } e, \texttt{status } s, \texttt{refines } e_1, \ldots, e_n, \cdots \texttt{end}]\!] = \{[\![e_1]\!], \ldots, [\![e_n]\!]\}$

  $ref\ [\![e_{init}]\!] = \{[\![e_{init}]\!]\}$

Figure 4.4: The semantics of $\mathcal{EVT}$-signature extraction for machines.

- $\mathbb{D} : Context \to Env \to Env$ *# Extract and store the signature for a context*

$$\mathbb{D}\left[\!\left[\begin{array}{l}\texttt{context } ctx \\ \texttt{extends } ctx_1, \ldots, ctx_n \\ cbody \\ \texttt{end}\end{array}\right]\!\right] \xi \;=\; \xi \cup \{[\![ctx]\!] \mapsto \begin{array}{l}(\mathbb{D}[\![cbody]\!] \;\cup \\ \;\xi[\![ctx_1]\!] \;\cup\; \ldots \;\cup\; \xi[\![ctx_n]\!])\}\end{array}$$

- $\mathbb{D} : ContextBody \to |\mathbf{Sign}_{\mathcal{FOPEQ}}|$ *# Extract the $\mathcal{FOPEQ}$ signature from a context*

$$\mathbb{D}\left[\!\left[\begin{array}{l}\texttt{sets } s_1, \ldots, s_n \\ \texttt{constants } c_1, \ldots, c_n \\ \texttt{axioms } a_1, \ldots, a_n \\ \texttt{theorems } t_1, \ldots, t_n\end{array}\right]\!\right] = \langle S, \Omega, \Pi \rangle \qquad \text{\# Sorts, operations, predicates}$$

where

$$\langle S, \Omega, \Pi \rangle = \mathbb{M}(([\![s_1]\!], \ldots, [\![s_n]\!]), ([\![c_1]\!], \ldots, [\![c_n]\!]), ([\![a_1]\!], \ldots, [\![a_m]\!]))$$

Figure 4.5: The semantics of $\mathcal{FOPEQ}$-signature extraction uses the interface described in Figure 4.3 in order to extract signature components from the definition of a *ContextBody*.

```
1  Σmac2 = ⟨ S, Ω, Π, E, V ⟩
2    where
3    S = {Bool, COLOUR},
4    Ω = {red, green, orange},
5    Π = {},
6    E = {⟨Initialisation, ordinary⟩, ⟨set_cars_go, ordinary⟩, ⟨set_cars_stop,ordinary⟩,
7         ⟨set_peds_go, ordinary⟩, ⟨set_peds_stop, ordinary⟩, ⟨set_cars_green, ordinary⟩,
8         ⟨set_cars_red, ordinary⟩,⟨set_peds_red, ordinary⟩,⟨set_peds_green, ordinary⟩,
9         ⟨press_button, ordinary⟩},
10   V = {cars_go:Bool, peds_go:Bool, cars_colour:COLOUR, peds_colour:COLOUR, button_pushed : Bool}
```

Figure 4.6: Signature extracted by application of the semantic functions in Figures 4.4 and 4.5 to the Event-B machine specification of `mac2` in Figure 2.1.

In order to illustrate this signature extraction in practice, Figure 4.6 contains the signature that was extracted, using these semantic functions from `mac2` in Figure 2.1. Notice how the variables and events in the signature include those from the abstract machine, `mac1` in Figure 2.1 (lines 1–21), and the constants from the context in Figure 2.1 (lines 22–27) have been included as 0-ary operators using the $\mathcal{FOPEQ}$ interface.

Once the environment has been formed, we can then define a systematic translation from specifications in Event-B to presentations over $\mathcal{EVT}$.

We take a top-down approach to this translation which is comprised of two parts.

- The first semantic mapping (in Figures 4.7 and 4.8) that we provide is from the superstructure components of an Event-B specification to presentations over $\mathcal{EVT}$ (for machines) and presentations over $\mathcal{FOPEQ}$ (for contexts) in Section 4.5.
- The second semantic mapping (in Figures 4.9 and 4.10) that we define is from the Event-B infrastructure sentences (invariants, variants, events and axioms) to sentences over $\mathcal{EVT}$ (for invariants, variants and events) and sentences over $\mathcal{FOPEQ}$ (for axioms) in Section 4.6.

4.5 DEFINING THE SEMANTICS OF EVENT-B SUPERSTRUCTURE SENTENCES

Based on the syntax defined in Figure 4.2 we have identified the constructs that form the Event-B superstructure language:

- `extends` *context_identifier*$^+$
- `refines` *machine_identifier*
- `sees` *context_identifier*$^+$
- `refines` *event_identifier*$^+$

In this section, we define a semantics for the Event-B superstructure language using specification- building operators. In Figures 4.7 and 4.8 we define the semantic function $\mathbb{B}$ to translate Event-B specifications written using the superstructure language to presentations over $\mathcal{EVT}$ that use the specification-building operators defined in the theory of institutions [100]. We translate a specification as described by Figure 4.2 into a presentation over the institution $\mathcal{EVT}$.

The semantics of an Event-B specification SP are given by $\mathbb{B}[\![SP]\!]\xi$, *where* $\xi = \mathbb{D}[\![SP]\!]\xi_0$ *is the environment defined by Figure 4.4.*

- $\mathbb{B} : \mathit{Specification} \to \mathit{Env} \to \langle|\mathbf{Pres}|\rangle$ *# Process specifications in an environment, build presentations*

  $\mathbb{B}\;[\![\langle\rangle]\!]\;\xi \;=\; \langle\rangle$

  $\mathbb{B}\;[\![hd :: tl]\!]\;\xi \;=\; (\mathbb{B}\;[\![hd]\!]\;\xi) :: (\mathbb{B}\;[\![tl]\!]\;\xi)$

- $\mathbb{B} : \mathit{Machine} \to \mathit{Env} \to |\mathbf{Pres}_{\mathcal{EVT}}|$ *# Build an* $\mathcal{EVT}$ *presentation for a machine*

$$\mathbb{B}\left[\!\!\left[\begin{array}{l}\texttt{machine}\ m\\ \texttt{refines}\ a\\ \texttt{sees}\ ctx_1,\ldots,ctx_n\\ mbody\\ \texttt{end}\end{array}\right]\!\!\right]\xi \;=\; \left\langle \Sigma, \left[\begin{array}{l}\texttt{spec}\ [\![m]\!]\ \texttt{over}\ \mathcal{EVT} =\\ \quad([\![ctx_1]\!]\ \texttt{and}\ \ldots\ \texttt{and}\ [\![ctx_n]\!])\\ \quad\quad \texttt{with}\ \rho\\ \quad(\texttt{and}\ \mathbb{A}_\Sigma[\![mbody]\!][\![a]\!]\xi)^*\\ \texttt{then}\\ \quad\mathbb{S}_\Sigma[\![mbody]\!]\\ where\\ \quad\rho : \mathcal{FOPEQ} \to \mathcal{EVT}\end{array}\right]\right\rangle$$

where $\Sigma = \xi[\![m]\!]$. $^*\mathbb{A}_\Sigma$ *is only used if the* `refines` *clause is nonempty*

- $\mathbb{A}_\Sigma : \mathit{MachineBody} \to \mathit{identifier} \to \mathit{Env} \to \mathit{Sen}(\Sigma)$ *# Add relevant sentences from the abstract machine*

$$\mathbb{A}_\Sigma\left[\!\!\left[\begin{array}{l}\texttt{variables}\ v_1,\ldots,v_n\\ \texttt{invariants}\ i_1,\ldots,i_n\\ \texttt{theorems}\ t_1,\ldots,t_n\\ \texttt{variant}\ n\\ \texttt{events}\ e_{init},\ e_1,\ldots,e_n\end{array}\right]\!\!\right][\![a]\!]\;\xi \;=\; \begin{array}{l}\mathbb{I}_\Sigma[\![i_1]\!]\ \texttt{and}\ \ldots\ \texttt{and}\ \mathbb{I}_\Sigma[\![i_n]\!]\\ \texttt{and}\ \mathbb{R}_\Sigma[\![e_1]\!][\![a]\!]\xi\\ \texttt{and}\ \ldots\ \texttt{and}\ \mathbb{R}_\Sigma[\![e_n]\!][\![a]\!]\xi\end{array}$$

 # Conjoin sentences from each event definition

Figure 4.7: The semantics of Event-B superstructure sentences are defined by translating them into presentations over $\mathcal{EVT}$ using the semantic function $\mathbb{B}$ and the specification-building operators defined in the theory of institutions (Table 2.3). Recall from Definition 29 that objects of **Pres** are of the form $\langle\Sigma, \Phi\rangle$ for a signature Σ and $\Phi \subseteq \mathit{Sen}(\Sigma)$. This figure contains the translation for machine specifications, the translation of events and contexts is outlined in Figure 4.8.

- $\mathbb{R}_\Sigma : Event \to identifier \to Env \to Sen(\Sigma)$ *# Get sentences from each abstract refined event*

$$\mathbb{R}_\Sigma \left[\!\!\left[\begin{array}{l} \texttt{event } e_c \\ \texttt{status } s \\ \texttt{refines } e_1,\ldots,e_n \\ ebody \\ \texttt{end} \end{array} \right]\!\!\right] [\![a]\!]\ \xi \ =$$

```
let
```
$\Sigma_a = \xi[\![a]\!]$,

 # Signature of abstract machine

$\Sigma_h = \langle \Sigma_a.S, \Sigma_a.\Omega, \Sigma_a.\Pi, \{[\![e_1]\!], \ldots, [\![e_n]\!]\} \lhd \Sigma_a.E, \Sigma_a.V \rangle$,

$\sigma_h : \Sigma_h \hookrightarrow \Sigma_a$,

 # Keep only refined (abstract) events

$\sigma_m : \Sigma_h \to \Sigma$

 # Reassign refined event sentences to e_c

$$\sigma_m = \left\langle \begin{array}{l} \Sigma_h.S \hookrightarrow \Sigma.S,\ \Sigma_h.\Omega \hookrightarrow \Sigma.\Omega, \\ \Sigma_h.\Pi \hookrightarrow \Sigma.\Pi, \\ \Sigma_h.E \mapsto \{[\![e_c]\!]\} \lhd \Sigma.E, \\ \Sigma_h.V \hookrightarrow \Sigma.V \end{array} \right\rangle$$

```
in
```
$([\![a]\!]$ `hide via` $\sigma_h)$ `with` σ_m.

- $\mathbb{B} : \ Context \to Env \to |\mathbf{Pres}_{\mathcal{FOPEQ}}|$ *# N.B. We get a $\mathcal{FOPEQ}$ presentation for contexts*

$$\mathbb{B} \left[\!\!\left[\begin{array}{l} \texttt{context } ctx \\ \texttt{extends } ctx_1,\ldots,ctx_n \\ cbody \\ \texttt{end} \end{array} \right]\!\!\right] \xi \ = \left\langle \Sigma, \left[\begin{array}{l} \texttt{spec } [\![ctx]\!] \texttt{ over } \mathcal{FOPEQ} = \\ \quad [\![ctx_1]\!] \texttt{ and } \ldots \texttt{ and } [\![ctx_n]\!] \\ \texttt{then} \\ \quad \mathbb{S}_\Sigma [\![cbody]\!] \end{array} \right] \right\rangle$$

where $\Sigma = \xi[\![ctx]\!]$.

Figure 4.8: The translation of the event and context components of the Event-B superstructure sentences. This is a continuation of the translation described in Figure 4.7.

- $\mathbb{S}_\Sigma : \mathit{MachineBody} \to \mathit{Sen}_{\mathcal{EVT}}(\Sigma)$ *# Build sentences from a machine body*

$$\mathbb{S}_\Sigma \left[\!\!\left[\begin{array}{l} \texttt{variables } v_1,\ldots,v_n \\ \texttt{invariants } i_1,\ldots,i_n \\ \texttt{theorems } t_1,\ldots,t_n \\ \texttt{variant } n \\ \texttt{events } e_{init},\, e_1,\ldots,e_n \end{array} \right]\!\!\right] = \left(\begin{array}{ll} & \mathbb{I}_\Sigma[\![i_1]\!] \cup \ldots \cup \mathbb{I}_\Sigma[\![i_n]\!] \\ \cup & \mathbb{V}_\Sigma[\![n]\!] \\ \cup & \mathbb{E}_\Sigma[\![e_{init}]\!] \\ \cup & \mathbb{E}_\Sigma[\![e_1]\!] \cup \ldots \cup \mathbb{E}_\Sigma[\![e_n]\!] \end{array} \right)$$

- $\mathbb{I}_\Sigma : \mathit{LabelledPred} \to \mathit{Sen}_{\mathcal{EVT}}(\Sigma)$ *# Invariants*

$$\mathbb{I}_\Sigma \quad [\![i]\!] = \{\langle [\![inv]\!],\ \texttt{F.and}(\mathbb{P}_\Sigma[\![i]\!], \texttt{F.}\iota(\Sigma.V)(\mathbb{P}_\Sigma[\![i]\!]))\rangle\}$$

- $\mathbb{V}_\Sigma : \mathit{expression} \to \mathit{Sen}_{\mathcal{EVT}}(\Sigma)$ *# Variant can't increase for non-ord. events*

$$\begin{array}{rl} \mathbb{V}_\Sigma \quad [\![n]\!] = & \{\langle [\![e]\!],\ \texttt{F.lt}(\texttt{F.}\iota(\Sigma.V)(\mathbb{T}_\Sigma[\![n]\!]), \mathbb{T}_\Sigma[\![n]\!])\rangle \mid (e, \texttt{convergent}) \in E\} \\ \cup & \{\langle [\![e]\!],\ \texttt{F.leq}(\texttt{F.}\iota(\Sigma.V)(\mathbb{T}_\Sigma[\![n]\!]), \mathbb{T}_\Sigma[\![n]\!])\rangle \mid (e, \texttt{anticipated}) \in E\} \end{array}$$

- $\mathbb{E}_\Sigma : \mathit{InitEvent} \to \mathit{Sen}_{\mathcal{EVT}}(\Sigma)$ *# Initial event: get sentences from actions*

$$\mathbb{E}_\Sigma : \left[\!\!\left[\begin{array}{l} \texttt{event Initialisation} \\ \texttt{status ordinary} \\ \texttt{then } act_1,\ldots,act_n \\ \texttt{end} \end{array} \right]\!\!\right] = \{\langle \texttt{Initialisation},\ BA\rangle\}$$

where

$$BA = \texttt{F.and}(\mathbb{P}_\Sigma[\![act_1]\!], \ldots, \mathbb{P}_\Sigma[\![act_n]\!])$$

Figure 4.9: A semantics for Event-B infrastructure sentences is provided by translating them into sentences over $\mathcal{EVT}$, denoted $\mathit{Sen}_{\mathcal{EVT}}(\Sigma)$, for machines and sentences over $\mathcal{FOPEQ}$, denoted $\mathit{Sen}_{\mathcal{FOPEQ}}(\Sigma)$, for contexts. We use the interface operations and semantic functions described in Figure 4.3 throughout this translation. The event and context components of this translation are contained in Figure 4.10.

- $\mathbb{E}_\Sigma : \mathit{Event} \rightarrow \mathit{Sen}_{\mathcal{EVT}}(\Sigma)$ # *Non-initial event: get sentences from event body*

$$\mathbb{E}_\Sigma : \left[\!\left[\begin{array}{l} \texttt{event } e \\ \texttt{status } s \\ \texttt{refines } e_1, \ldots, e_n \\ \mathit{ebody} \\ \texttt{end} \end{array} \right]\!\right] = \{\langle [\![e]\!],\ \mathbb{F}_\Sigma[\![\mathit{ebody}]\!]\rangle\}$$

- $\mathbb{F}_\Sigma : \mathit{EventBody} \rightarrow \Sigma\text{-}\mathit{formula}$ # *Build a formula for an event definition*

$$\mathbb{F}_\Sigma : \left[\!\left[\begin{array}{l} \texttt{any } p_1, \ldots, p_n \\ \texttt{where } grd_1, \ldots, grd_n \\ \texttt{with } w_1, \ldots, w_n \\ \texttt{then } act_1, \ldots, act_n \end{array} \right]\!\right] = \texttt{F.exists}(p,\ \texttt{F.and}(G, W, BA))$$

Formula is existentially quantified over event parameters p

where

$p = \langle [\![p_1]\!], \ldots, [\![p_n]\!] \rangle$ # *List of parameters*

$G = \texttt{F.and}(\mathbb{P}_\Sigma[\![grd_1]\!], \ldots, \mathbb{P}_\Sigma[\![grd_n]\!])$ # *Guards*

$W = \texttt{F.and}(\mathbb{P}_\Sigma[\![w_1]\!], \ldots, \mathbb{P}_\Sigma[\![w_n]\!])$ # *Witnesses*

$BA = \texttt{F.and}(\mathbb{P}_\Sigma[\![act_1]\!], \ldots, \mathbb{P}_\Sigma[\![act_n]\!])$ # *Actions*

- $\mathbb{S}_\Sigma : \mathit{ContextBody} \rightarrow \mathit{Sen}_{\mathcal{FOPEQ}}(\Sigma)$ # *Context: get sentences from axioms*

$$\mathbb{S}_\Sigma \left[\!\left[\begin{array}{l} \texttt{sets } s_1, \ldots, s_n \\ \texttt{constants } c_1, \ldots, c_n \\ \texttt{axioms } a_1, \ldots, a_n \\ \texttt{theorems } t_1, \ldots, t_n \end{array} \right]\!\right] = \{\mathbb{P}_\Sigma[\![a_1]\!], \ldots, \mathbb{P}_\Sigma[\![a_n]\!]\}$$

Figure 4.10: This is a continuation of the Event-B infrastructure sentence translation outlined in Figure 4.9. Here, we provide the event and context specific components of the translation of the infrastructure sentences.

The construct that enables a context to extend others is used in Event-B to add more details to a context. Since a context in Event-B only refers to elements of the $\mathcal{FOPEQ}$ component of an $\mathcal{EVT}$ signature it is straightforward to translate this using the specification-building operator `then`. As outlined in Table 2.3, `then` is used to enrich the signature with new sorts/operations etc. [100].

A context can be extended by more than one context. In this case the resulting context contains all constants and axioms of all extended contexts and the additional specification of the extending context itself [6]. To give a semantics for context extension using the specification-building operators we `and` all extended contexts and use `then` to incorporate the extending context itself. The specification-building operator `and` takes the sum of two specifications that can be written over different signatures (Table 2.3). It is the most straightforward way to combine specifications over different signatures [100].

In Event-B, a machine **SEES** one or more contexts. This construct is used to add a context(s) to a machine so that the machine can refer to elements of the context(s). From Theorem 32, we know that the relationship between $\mathcal{FOPEQ}$ and $\mathcal{EVT}$ is that of an institution comorphism. This enables us to directly use $\mathcal{FOPEQ}$-sentences, as given by the context in this case, in an $\mathcal{EVT}$-presentation. We use the specification-building operation `with` ρ which indicates translation by an institution comorphism ρ [100]. The resulting machine specification is heterogeneous as it links two institutions, $\mathcal{EVT}$ and $\mathcal{FOPEQ}$, by the institution comorphism described in Definition 38 and Theorem 32.

An Event-B machine can refine at most one other machine and, as previously outlined in Section 2.1.1, there are two types of machine refinement: superposition and data refinement [6]. The specification-building operator, `then`, can account for both of these types of refinement because either new signature components or constraints on the

data (gluing invariants) are added to the specification. In Figure 4.7 the semantic function $\mathbb{A}_\Sigma$ is used to process the events in the concrete machine which refine those in the abstract machine.

Event refinement in Event-B is superposition refinement [6]. By superposition refinement all of the components of the corresponding abstract event are implicitly included in the refined version. This approach is useful for gradually adding more detail to the event. In $\mathcal{EVT}$, we have not prohibited multiple definitions of the same event name. When there are multiple definitions we combine them by taking the conjunction of their respective formulae which will constrain the model. As mentioned above, when refining an abstract machine we use the semantic function $\mathbb{A}_\Sigma$, in Figure 4.7 to process the events in the concrete machine which refine those in the abstract machine. $\mathbb{A}_\Sigma$ in turn calls the semantic function $\mathbb{R}_\Sigma$ (Figure 4.8).

$\mathbb{R}_\Sigma$ restricts the event component of the abstract machine signature to those events contained in the `refines` clause of the event definition using domain restriction ($\lhd$). $\mathbb{A}_\Sigma$ also extracts the invariant sentences from the abstract machine. This new signature is included in the abstract via the signature morphism σ_h. We then form the signature morphism σ_m which is the identity on the sort, operation, predicate and variable components of Σ_h. σ_m maps each of the abstract event signature components that are being refined by the concrete event e_c to the signature component corresponding to e_c. The resulting sentence uses `hide via` and `with` to apply these signature morphisms (σ_h and σ_m) correctly.

Figures 4.7 and 4.8 describe our translational semantics for the Event-B superstructure sentences, next, we show how it is applied for the Event-B infrastructure sentences.

4.6 DEFINING THE SEMANTICS OF EVENT-B INFRASTRUCTURE SENTENCES

In this section, we define a translation from Event-B infrastructure sentences to sentences over $\mathcal{EVT}$. We translate the axiom sentences that are found in Event-B contexts to sentences over $\mathcal{FOPEQ}$ as they form part of the underlying Event-B mathematical language as shown in Figure 4.1. We refer the reader to Section 3.3, where we described how to translate Event-B variants, invariants and events into $\mathcal{EVT}$-sentences, and our translational semantics described here mirrors this translation.

We define an overloaded meaning function, $\mathbb{S}_\Sigma$, for specifications in Figures 4.9 and 4.10. $\mathbb{S}_\Sigma$ takes as input a specification and returns a set of sentences over $\mathcal{EVT}$ ($Sen_{\mathcal{EVT}}(\Sigma)$) for machines and a set of sentences over $\mathcal{FOPEQ}$ ($Sen_{\mathcal{FOPEQ}}(\Sigma)$) for contexts. When applying $\mathbb{S}_\Sigma$ to a machine (resp. context) we also define semantic functions for processing invariants, variants and events (resp. axioms). These are given by $\mathbb{I}_\Sigma, \mathbb{V}_\Sigma$ and $\mathbb{E}_\Sigma$. Axioms are predicates that can be translated into closed $\mathcal{FOPEQ}$-formulae using the semantic function $\mathbb{P}_\Sigma$ which is defined in the interface in Figure 4.3.

Given a list of invariants $i_1, \ldots, i_n$ we define the semantic function $\mathbb{I}_\Sigma$ in Figure 4.9. Each invariant, i, is a *LabelledPred* from which we form the open $\mathcal{FOPEQ}$-sentence $\texttt{F.and}(\mathbb{P}_\Sigma[\![i]\!], \texttt{F.}\iota(\Sigma.V)(\mathbb{P}_\Sigma[\![i]\!]))$. Each invariant sentence is paired with the invariant identifier *inv* to ensure that the invariant is applied to all events when evaluating the satisfaction condition.

Given a variant expression n, we define the semantic function $\mathbb{V}_\Sigma$ in Figure 4.9. The variant is only relevant for specific events so we pair it with an event name in order to meaningfully evaluate the variant expression. As described in Section 2.1.1, an event whose status is `convergent` must strictly decrease the variant expression. An event whose status is

`anticipated` must not increase the variant expression. This expression can be translated into an open $\mathcal{FOPEQ}$-term using the semantic function $\mathbb{T}_\Sigma$ as described in Figure 4.3. From this we form a formula based on the status of the event(s) in the signature Σ as described in Section 3.3. Event-B machines are only permitted to have one variant [6].

In Figure 4.10 we define the semantic function $\mathbb{E}_\Sigma$ to process a given event definition. Event guard(s) and witnesses are predicates that can be translated via $\mathbb{P}_\Sigma$ into open $\mathcal{FOPEQ}$-formulae denoted by G and W respectively in Figure 4.10. In Event-B, actions are interpreted as before-after predicates e.g. `x := x + 1` is interpreted as `x' = x + 1`. Therefore, actions can also be translated via $\mathbb{P}_\Sigma$ into open $\mathcal{FOPEQ}$-formulae denoted by BA in Figure 4.10. Thus the semantics of an *EventBody* definition is given by the semantic function $\mathbb{F}_\Sigma$ which returns the formula $\texttt{F.exists}(p, \texttt{F.and}(G, W, BA))$ where p is a list of the event parameters.

A context can exist independently of a machine and is written as a specification over $\mathcal{FOPEQ}$. Thus, we translate an axiom sentence directly as a $\mathcal{FOPEQ}$-sentence which is a closed Σ-formula using the semantic function $\mathbb{P}_\Sigma$ given in Figure 4.3. Axiom sentences are closed $\mathcal{FOPEQ}$-formulae (elements of $Sen_{\mathcal{FOPEQ}}(\Sigma)$) which are interpreted as a valid sentences in $\mathcal{EVT}$ using the comorphism ρ.

Thus, we have formalised an institution-based translational semantics for the three layers of the Event-B language. We illustrate this semantics via an example in the next section.

4.7 IMPLEMENTING THE TRANSLATIONAL SEMANTICS

In order to validate our translational semantics we developed a Haskell [118] translator that generates the associated $\mathcal{EVT}$-specifications when given Event-B specifications, called `EB2EVT`. Event-B machines and contexts are stored as XML files (`.bum` for machine files and `.buc` for context

```
1 bbEventDefines ::[(Ident, Ident)]-> [EventDef] -> [Sentence]
2 bbEventDefines xx' evtList =
3   let pp eBody = (eany eBody)
4       gg eBody = F.and (ewhere eBody)
5       ww eBody = F.and (ewith eBody)
6       ba eBody = F.and (ethen eBody)
7       bbEvent eBody = (F.forall (xx' ++(pp eBody)) (F.and [gg eBody, ww eBody, ba eBody]))
8       bbEventDef evt = (Sentence (ename evt) (bbEvent (ebody evt)))
9   in (map bbEventDef evtList)
```

Figure 4.11: The Haskell implementation of the $\mathbb{B}$ function from Figure 4.9 as applied to a list of Event-B event definitions.

files) and we parse these to generate the associated $\mathcal{EVT}$-specifications. Our parser uses an Abstract Syntax Tree (AST) to internally represent the Event-B specifications and the $\mathcal{EVT}$-specifications are generated by traversing this AST[1]. The generated $\mathcal{EVT}$-specifications are syntactically sugared in a Hets-like notation.

The source code for `EB2EVT` consists of four Haskell files resulting in approximately 700 lines of code and is available at `https://github.com/mariefarrell/phdartefacts.git`. We provide implementations of the syntactic definitions and semantic functions that are contained in Figures 4.2 (`Syntax.hs`), 4.3 (`FOPEQ.hs`), 4.4, 4.5, 4.7, 4.8, 4.9 and 4.10 (`Semantics.hs`). In Figure 4.11 we provide the associated code snippet of the $\mathbb{B}$ function from Figure 4.9. The reader need only refer back to Figure 4.9 to see their correspondence.

The file called `ParseEB.hs` parses the raw XML input to an AST and returns the corresponding $\mathcal{EVT}$-specifications using the syntactic definitions and semantic functions that we have constructed.

4.8 Applying the Translational Semantics to an Example

We illustrate the use of `EB2EVT` using the cars on a bridge example presented in Abrial's book [3, Ch. 2]. This Event-B specification has three

1 More details on the implementation of `EB2EVT` can be found in Appendix B.4.2.

```
1 CONTEXT cd
2   CONSTANTS
3     d
4   AXIOMS
5      axm1: d ∈ ℕ
6      axm2: d > 0
7 END
```

```
8  MACHINE m0
9    SEES cd
10   VARIABLES
11     n
12   INVARIANTS
13      inv1: n ∈ ℕ
14      inv2: n ⩽ d
15      inv3: n < 0 ∨ n < d
16   EVENTS
17     Initialisation
18       then
19          act1: n := 0
20     Event ML_out ≙ordinary
21       when
22          grd1: n < d
23       then
24          act1: n := n+1
25      Event ML_in ≙ordinary
26        when
27           grd1: n > 0
28        then
29           act1: n := n − 1
30 END
```

Figure 4.12: Event-B abstract machine m0 cd. This specification consists of the events ML_out and ML_in that model the behaviour of cars leaving and entering the mainland respectively.

refinement steps, resulting in four machines and three contexts. The final refinement step results in quite a large specification and so we omit it here but it can be found in Appendix B.4.4. We present the resultant $\mathcal{EVT}$-specification corresponding to each of the first two refinement steps. We also provide an independent, modular version of this specification in Appendix B.4.4 to illustrate some alternative approaches to modularisation for this example.

4.8.1 THE ABSTRACT MODEL

Figure 4.12 contains an Event-B specification corresponding to the first abstract machine in this development. It describes the behaviour of cars entering and leaving the mainland. This abstract model is comprised of a context (lines 1–7) specifying a natural number constant and a basic

```
1 spec  cd =
2   ops  d:ℕ
3   .  d > 0
4 end
```

```
6 spec  m0 =
7   cd
8   then
9     ops  n:ℕ
10    .  n⩽ d
11       n > 0 ∨ n<d
12    Events
13      Initialisation
14        thenAct  n := 0
15      Event ML_out ≙ordinary
16        when  n<d
17        thenAct  n:=n+1
18      Event ML_in ≙ordinary
19        when  n>0
20        thenAct  n:=n-1
21 end
```

Figure 4.13: Syntactically sugared $\mathcal{EVT}$-specification as generated by EB2EVT that corresponds to the Event-B model in Figure 4.12.

machine description (lines 8–30). The corresponding $\mathcal{EVT}$-specification generated by EB2EVT is presented in Figure 4.13.

LINES 1–4: This is a $\mathcal{CASL}$-specification that describes the context from Figure 4.12. The constant d is represented as an operation of the appropriate type and the non-typing axioms are included as predicates. This is achieved by via the semantic function $\mathbb{P}$ that was defined in the $\mathcal{FOPEQ}$ interface (Figure 4.3).

LINES 5–20: This is the $\mathcal{EVT}$-specification that is generated corresponding to the abstract machine in Figure 4.12 (lines 8–30). The machine variable, n is represented as an operation and the context specification is included using then on lines 6 and 7. This corresponds to the application of the semantic function $\mathbb{B}$ (Figures 4.7 and 4.8) to the superstructure components of the machine.

With regard to the infrastructure translation described in Figures 4.9 and 4.10, note that the invariant and event sentences have been syntactically sugared in order to simplify the notation.

4.8.2 THE FIRST REFINEMENT

In the first refinement step, as can be seen in Figure 4.14, new events are added to describe cars entering and leaving the island (`IL_out` and `IL_in` on lines 30–42). New variables are added to record the number of cars on the island, cars on the mainland and those on the bridge. The events `ML_in` and `ML_out` are also refined to utilise the new variables, `a`, `b` and `c`, that are introduced on line 5. Figure 4.15 contains an $\mathcal{EVT}$-specification corresponding to this Event-B machine specification.

LINES 1–3: The application of $\mathbb{B}$ to the `refines` and **SEES** clauses of Figure 4.14 (lines 2–3) results in the application of the specification-building operators (`and`, `then`) to include the abstract machine and context specifications into the generated $\mathcal{EVT}$-specification.

LINES 4–9: The new variables and (non-theorem) invariants are included as operations and predicates respectively.

LINES 10–32: $\mathbb{R}_\Sigma$ returns (`m`o `hide via` σ_h) `with` σ_m for each refined event. The event names have remained the same during event refinement in this example, and since events with the same name are implicitly merged we can omit this notation here.

4.8.3 THE SECOND REFINEMENT

The second refinement step (Figure 4.16) adds a lot of new behaviour to the Event-B model. The first addition is a context (lines 1–8) containing colours which provide values for the new variables (`ml_tl` and `il_tl`). These are used to control the behaviour of a pair of traffic lights (line 14). The new variables `ml_pass` and `il_pass` are used to record whether or not a car has passed through the mainland or island traffic light

```
1  MACHINE m1
2    refines m0
3    SEES cd
4    VARIABLES
5      a, b, c
6    INVARIANTS
7       inv1: a ∈ ℕ
8       inv2: b ∈ ℕ
9       inv3: c ∈ ℕ
10      inv4: n = a + b + c
11      inv5: a = 0 ∨ c = 0
12      thm1: a + b + c ∈ ℕ  theorem
13      thm2: c > 0 ∨ a > 0
14            ∨ (a + b < d ∧ c = 0)
15            ∨ (0 < b ∧ a = 0) theorem
16   VARIANT 2*a + b
17   EVENTS
18     Initialisation
19       then
20          act2: a := 0
21          act3: b := 0
22          act4: c := 0
23   Event ML_out ≙ordinary
24     refines ML_out
25     when
26        grd1: a + b < d
27        grd2: c = 0
28     then
29        act1: a := a + 1
30   Event IL_in ≙convergent
31     when
32        grd1: a > 0
33     then
34        act1: a := a − 1
35        act2: b := b + 1
36   Event IL_out ≙convergent
37     when
38        grd1: 0 < b
39        grd2: a = 0
40     then
41        act1: b := b − 1
42        act2: c := c + 1
43   Event ML_in ≙ordinary
44     refines ML_in
45     when
46        grd1: c > 0
47     then
48        act2: c := c − 1
49 END
```

Figure 4.14: Event-B machine `m1` with additional events `IL_in` and `IL_out` to model the behaviour of cars entering and leaving the island. The variables `a`, `b`, and `c` keep track of the number of cars on the bridge going to the island, the number of cars on the mainland and the number of cars on the bridge going to the mainland respectively.

respectively. The objective of these variables is to ensure that the traffic lights do not change so rapidly that no car can pass through them [3]. Events for these lights were added to the machine and the current events were modified to account for this behaviour. In particular each of the abstract events (`ML_out` and `IL_out` in Figure 4.14) are refined by two events (`ML_out1`, `ML_out2`, `IL_out1` and `IL_out2` in Figure 4.16).

LINES 1–7: This $\mathcal{CASL}$ specification specifies the new data type `Color`. This corresponds to the context described in the Event-B model in Figure 4.16.

```
1  spec  m1 =
2    m0 and cd
3      then
4      ops   a:ℕ
5            b:ℕ
6            c:ℕ
7      .  n=a+b+c
8         a=0 ∨ c=0
9      variant  2*a+b
10     EVENTS
11       Initialisation
12         thenAct   a := 0
13                   b := 0
14                   c := 0
15       Event ML_out ≙ordinary
16         when  a+b<d
17               c=0
18         thenAct  a:=a+1
19       Event IL_in ≙convergent
20         when  a>0
21         thenAct a:=a-1
22                 b:=b+1
23       Event IL_out ≙convergent
24         when  0<b
25               a=0
26         thenAct  b:=b-1
27                  c:=c+1
28       Event ML_in ≙ordinary
29         when  c>0
30         thenAct  c:=c-1
31 end
```

Figure 4.15: $\mathcal{EVT}$-specification corresponding to the first refinement step as presented in Figure 4.14.

LINES 8–14: The superstructure component of the translation includes the abstract machine (`m0`), and the contexts (`cd` and `COLOR`). In this case the names of the events are changed in the refinement step. The event `ML_out` is decomposed into the events `Ml_out1` and `ML_out2`. The event `IL_out` follows a similar decomposition into `IL_out1` and `IL_out2`. We account for this refinement using the signature morphisms and specification-building operators on lines 10–13. We have not explicitly described these signature morphisms but their functionality should be obvious from their respective subscripts.

LINE 26: We use a simple variant notation for the variant expression.

LINES 27–83: The events are translated as before for the remainder of the specification.

This example illustrates the use of EB2EVT as a means for bridging the gap between the Rodin and HETS software eco-systems.

```
1   CONTEXT Color
2     SETS Color
3     CONSTANTS
4       red, green
5     AXIOMS
6         axm4: Color = {green, red}
7         axm3: green ≠ red
8   END

9   MACHINE m2
10    refines m1
11    SEES cd, Color
12    VARIABLES
13      a, b, c,
14      ml_tl, il_tl,
15      il_pass, ml_pass
16    INVARIANTS
17      inv1: ml_tl ∈ {red, green}
18      inv2: il_tl ∈ {red, green}
19      inv3: ml_tl = green ⇒ c = 0
20      inv12: ml_tl = green ⇒ a + b + c < d
21      inv4: il_tl = green ⇒ a = 0
22      inv11: il_tl = green ⇒ b > 0
23      inv6: il_pass ∈ {0, 1}
24      inv7: ml_pass ∈ {0, 1}
25      inv8: ml_tl = red ⇒ ml_pass = 1
26      inv9: il_tl = red ⇒ il_pass = 1
27      inv5: il_tl = red ∨ ml_tl = red
28      thm2: 0 ⩾ a ⇒ a = 0 theorem
29      thm3: 0 ⩾ b ⇒ b = 0 theorem
30      thm4: 0 ⩾ c ⇒ c = 0 theorem
31      thm5: ¬ (d ⩽ 0) theorem
32      thm6: b + 1 ⩾ d ∧ ¬ (b + 1 = d)
33          ⇒ ¬ (b < d) theorem
34      thm7: b ⩽ 1 ∧ ¬ (b = 1)
35          ⇒ ¬ (b > 0) theorem
36      thm1: (ml_tl = green ∧ a + b + 1 < d)
37            ∨ (ml_tl = green ∧ a + b + 1 = d)
38            ∨ (il_tl = green ∧ b > 1)
39            ∨ (il_tl = green ∧ b = 1)
40            ∨ (ml_tl = red ∧ a + b < d
41               ∧ c = 0 ∧ il_pass = 1)
42            ∨ (il_tl = red ∧ 0 < b ∧ a = 0
43               ∧ ml_pass = 1)
44            ∨ 0 < a ∨ 0 < c theorem
45    VARIANT ml_pass + il_pass
46    EVENTS
47      Initialisation
48        then
49          act2: a := 0
50          act3: b := 0
51          act4: c := 0
52          act1: ml_tl := red
53          act5: il_tl := red
54          act6: ml_pass := 1
55          act7: il_pass := 1
56      Event ML_out1 ≙ordinary
57        refines ML_out
58        when
59          grd1: ml_tl = green
60          grd2: a + b + 1 < d
61        then
62          act1: a := a + 1
63          act2: ml_pass := 1
64      Event ML_out2 ≙ordinary
65        refines ML_out
66        when
67          grd1: ml_tl = green
68          grd2: a + b + 1 = d
69        then
70          act1: a := a + 1
71          act2: ml_tl := red
72          act3: ml_pass := 1
73      Event IL_out1 ≙ordinary
74        refines IL_out
75        when
76          grd1: il_tl = green
77          grd2: b > 1
78        then
79          act1: b := b − 1
80          act2: c := c + 1
81          act3: il_pass := 1
82      Event IL_out2 ≙ordinary
83        refines IL_out
84        when
85          grd1: il_tl = green
86          grd2: b = 1
87        then
88          act1: b := b − 1
89          act2: il_tl := red
90          act3: c := c + 1
91          act4: il_pass := 1
92      Event ML_tl_green ≙convergent
93        when
94          grd1: ml_tl = red
95          grd2: a + b < d
96          grd3: c = 0
97          grd4: il_pass = 1
98        then
99          act1: ml_tl := green
100         act2: il_tl := red
101         act3: ml_pass := 0
102     Event IL_tl_green ≙convergent
103       when
104         grd1: il_tl = red
105         grd2: 0 < b
106         grd3: a = 0
107         grd4: ml_pass = 1
108       then
109         act1: il_tl := green
110         act2: ml_tl := red
111         act3: il_pass := 0
112     Event IL_in ≙ordinary
113       refines IL_in
114       when
115         grd11: 0 < a
116       then
117         act11: a := a − 1
118         act12: b := b + 1
119     Event ML_in ≙ordinary
120       refines ML_in
121       when
122         grd1: 0 < c
123       then
124         act1: c := c + 1
125 END
```

Figure 4.16: Event-B machine `m2` that refines the Event-B machine in Figure 4.14 by adding new events `Ml_tl_green` and `Il_tl_green`. The context `Color` on lines 1–8 adds a new data type which is used by the `ml_tl` and `il_tl` traffic light variables.

```
1  spec  COLOR =
2    sorts  Color
3    ops  red : Color
4          green : Color
5    .  Color =  {green,red }
6       green ≠ red
7  end

8  spec  m2 =
9    m1 and cd and COLOR and
10   (m1  hide via  σh ML_out ) with  σm ML_out↦ML_out1 and
11   (m1  hide via  σh ML_out ) with  σm ML_out↦ML_out2 and
12   (m1  hide via  σh IL_out ) with  σm IL_out↦IL_out1 and
13   (m1  hide via  σh IL_out ) with  σm IL_out↦IL_out2
14   then
15     ops  ml_tl : {red,green }
16       il_tl :  {red,green }
17       il_pass :  {0,1 }
18       ml_pass :  {0,1 }
19     .  ml_tl=green ⇒ c=0
20        ml_tl=green ⇒ a+b+c<d
21        il_tl=green ⇒ a=0
22        il_tl=green ⇒ b>0
23        ml_tl=red ⇒ ml_pass=1
24        il_tl=red ⇒ il_pass=1
25        il_tl=red ∨ ml_tl=red
26     variant  ml_pass+il_pass
27     EVENTS
28       Initialisation
29         thenAct  a := 0
30                  b := 0
31                  c := 0
32                  ml_tl := red
33                  il_tl := red
34                  ml_pass := 1
35                  il_pass := 1
36     Event ML_out1 ≙ordinary
37         when  ml_tl=green
38               a+b+1<d
39         thenAct  a := a+1
40                  ml_pass := 1
41     Event ML_out2 ≙ordinary
42         when  ml_tl=green
43               a+b+1=d
44         thenAct  a := a+1
45                  ml_tl := red
46                  ml_pass := 1
47     Event IL_out1 ≙ordinary
48         when  il_tl=green
49               b>1
50         thenAct  b := b-1
51                  c := c+1
52                  il_pass := 1
53     Event IL_out2 ≙ordinary
54         when  il_tl=green
55               b=1
56         thenAct  b := b-1
57                  il_tl := red
58                  c := c+1
59                  il_pass := 1
60     Event ML_tl_green ≙anticipated
61         when  ml_tl=red
62               a+b<d
63               c=0
64               il_pass=1
65         thenAct  ml_tl := green
66                  il_tl := red
67                  ml_pass := 0
68     Event IL_tl_green ≙anticipated
69         when  il_tl=red
70               0<b
71               a=0
72               ml_pass=1
73         thenAct  il_tl := green
74                  ml_tl := red
75                  il_pass := 0
76     Event IL_in ≙ordinary
77         when  0<a
78         thenAct  a := a - 1
79                  b := b+1
80     Event ML_in ≙ordinary
81         when  0<c
82         thenAct c := c - 1
83 end
```

Figure 4.17: $\mathcal{EVT}$-specification corresponding to the second refinement step as presented in Figure 4.16.

4.9 SUMMARY

In this chapter, we presented a three layer model for Event-B by decomposing it into its mathematical, infrastructure and superstructure sub-languages. We have formalised a translational semantics, using the theory of institutions, for each of these three constituent languages and thus for Event-B itself. We have implemented this semantics in the EB2EVT translator and shown how it generates syntactically sugared $\mathcal{EVT}$-specifications using the cars on a bridge example.

Our approach does not inhibit using multiple modelling domains to evaluate the semantics of an Event-B model. In fact, we have provided scope for the interoperability of Event-B with other formalisms that have been described in this framework and this will be discussed further in the chapters that follow.

Part II

INTEROPERABILITY

"Truth is invariant under change of notation."

– Goguen & Burstall

5

AN INSTITUTION-THEORETIC FOUNDATION FOR THE TRANSLATION FROM UML TO EVENT-B

In this chapter we present the institution-theoretic constructs required to facilitate interoperability between the Event-B specification language and the Unified Modelling Language (UML). In the preceding chapters, we used the theory of institutions to provide a mathematically sound framework, over which we defined a translational semantics for Event-B using the institutions for first-order predicate logic with equality ($\mathcal{FOPEQ}$) and Event-B ($\mathcal{EVT}$). We present an overview of (1) the existing institution for UML state machines, $\mathcal{UML}$, and (2) define the comorphism relationship between $\mathcal{UML}$ and $\mathcal{EVT}$, providing a generic basis for interoperability between both languages (represented as institutions). Our approach parallels that implemented in the UML-B plugin for Rodin *which translates state machines written in a UML-like language into Event-B [115]. We show how this translation provides a mathematical grounding for the UML-B* Rodin *plugin using our EB2EVT translator. In this way, we translate the Event-B specification that was generated by the UML-B plugin into an $\mathcal{EVT}$-specification. We show that this $\mathcal{EVT}$-specification matches the specification obtained from applying our institution comorphism to $\mathcal{UML}$.*

5.1 INTRODUCTION

Figure 5.1 frames our discussion by illustrating the various tools, languages and institutions at play throughout this chapter. We use the UML-B plugin in Rodin to model a state machine and then generate Event-B specifications. These Event-B specifications are then translated,

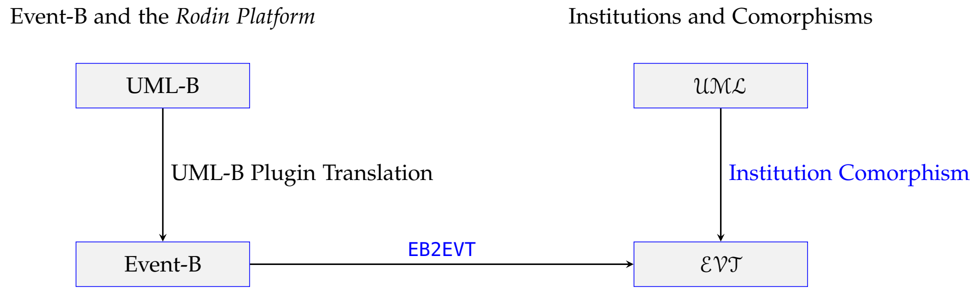


Figure 5.1: An overview of our approach to interoperability between UML and Event-B.

using EB2EVT, to $\mathcal{EVT}$-specifications. As an argument for correctness, we show that these $\mathcal{EVT}$-specifications correspond to the $\mathcal{EVT}$-specifications generated, using our comorphism, from the $\mathcal{UML}$ state machine.

The terminology of the institutions that we will utilise throughout this chapter is outlined in Table 5.1. This will become useful later as there is a substantial amount of notation used. In order to illustrate our constructions we will use the ATM machine example as outlined in Knapp et al. [73]. Figure 5.3 provides an illustration of the ATM behavioural state machine in question. We begin by describing the various components of the institution for simple UML machines, $\mathcal{UML}$. Then we define and prove the comorphism relationship between $\mathcal{UML}$ and $\mathcal{EVT}$. Finally, we illustrate the application of this comorphism to the ATM example and show how it provides a formal foundation for the UML-B plugin using EB2EVT.

5.2 $\mathcal{UML}$ - THE INSTITUTION FOR SIMPLE UML STATE MACHINES

The institution for UML state machines, denoted by $\mathcal{UML}$, that encompasses behavioural and protocol state machines has been previously defined by Knapp et al [73]. In this section, we review each of its components in detail. $\mathcal{UML}$ is made up of three separate logics/institutions as illustrated in Figure 5.2. The first being that for an OCL-like language

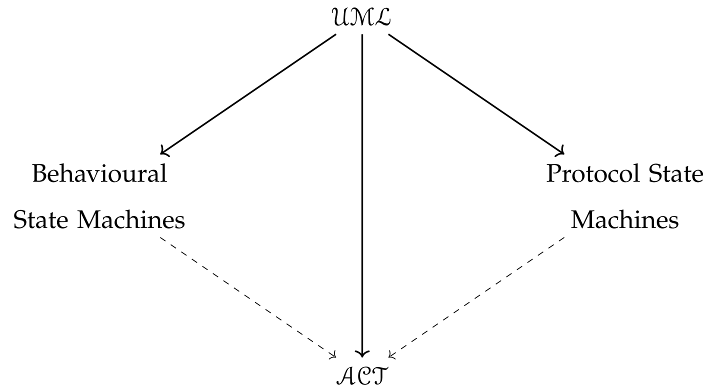


Figure 5.2: The state machine tripod of institutions, encompassing the institutions for behavioural and protocol state machines, is formed using an underlying institution of actions $\mathcal{ACT}$.

(denoted by $\mathcal{ACT}$), the second being an institution for behavioural state machines and the third component is a modification of the second to account for protocol state machines [73]. Knapp et al. use the running example on an ATM as illustrated in Figures 5.3 (behavioural state machine) and 5.9 (protocol state machine) to show that institution-theoretic refinement can capture the relationship between these state machines. The notation used for this institution is described in Table 5.1.

5.2.1 $\mathcal{ACT}$ - THE UNDERLYING ACTION INSTITUTION

Knapp et al. start by defining the action institution, $\mathcal{ACT}$, which is built over a primitive guard institution where sentences are guards (predicates) and models are variable-to-value mappings of the form $V_H \to Val$. The satisfaction condition of this guard institution is the evaluation of guard sentences using variable-to-value mappings given by the models. From our perspective, this corresponds to the institution for first-order predicate logic with equality, $\mathcal{FOPEQ}$ (Section 2.3.3), which forms the basis for our Event-B institution as defined in Chapter 3. The action institution is composed of the following:

Symbol	**Meaning**
$\mathcal{ACT}$	The institution for an OCL-like language. This is the underlying institution for $\mathcal{UML}$ (the action institution).
$H = \langle A_H, M_H, V_H \rangle$	Objects of the signature category of $\mathcal{ACT}$ where A_H is a set of actions (action statements), M_H is a set of messages and V_H is a set of sort-indexed variables.
$\Omega = \omega \xrightarrow[\Omega]{a,\overline{m}} \omega\prime$	Models of $\mathcal{ACT}$ are state transitions where ω and ω' are data states of the form $V_H \to Val$.
$\mathcal{UML}$	The institution for UML state machines as defined by Knapp et al. [73].
$\Sigma = \langle E_\Sigma, F_\Sigma, S_\Sigma \rangle$	Objects of the signature category of $\mathcal{UML}$ where E_Σ is a set of (external) events, F_Σ is a set of completion events and S_Σ is a set of states.
$\langle s_0, T \rangle$	Sentences written over $\mathcal{UML}$ where s_0 indicates an initial state. T represents a set of transitions of the form $s \xrightarrow[T]{p[g]/a,\bar{f}} s\prime$ where $p \in E_\Sigma \cup F_\Sigma$, g and a are guards and actions respectively written in the language of $\mathcal{ACT}$, and $\bar{f}$ is a set of completion events drawn from F_Σ.
$\langle I_\Theta, \Delta_\Theta \rangle$	Objects in the model category of $\mathcal{UML}$.
I_Θ	Initial configurations $I_\Theta \in \wp(V_H \to Val) \times S_\Sigma$.
Δ_Θ	Transition relation between configurations that emits messages. Each configuration consists of an action state ($V_H \to Val$), an event pool ($\wp(E_\Sigma \cup F_\Sigma)$) and a control state. The elements of Δ_Θ are of the form $(w, p, s) \xrightarrow{\overline{m}} (w\prime, p\prime, s\prime)$.
$\mathcal{EVT}$	The institution for Event-B.
$\Sigma = \langle S, \Omega, \Pi, E, V \rangle$	Objects of the signature category of $\mathcal{EVT}$ with $\langle S, \Omega, \Pi \rangle$ a signature over $\mathcal{FOPEQ}$. E is a set of (event name, status) pairs with *status* $\in$ {`convergent`, `anticipated`, `ordinary`} and V is a set of sort-indexed variable names.
$\langle e, \phi(\bar{x}, \bar{x}\prime) \rangle$	Sentences written over $\mathcal{EVT}$ where $e \in dom(E)$ and $\phi(\bar{x}, \bar{x}\prime)$ is a first-order formula whose free variables $\bar{x}, \bar{x}\prime$ are drawn from V and $V\prime$ respectively. Note that $V\prime$ is the same set as V but with all of the variable names primed.
$\langle A, L, R \rangle$	Objects of the model category of $\mathcal{EVT}$. A is a model over $\mathcal{FOPEQ}$, L is the initialising set of variable-to-value mappings and R is a set of event-indexed relations for each event name (except `Initialisation`) containing the before and the after variable-to-value mappings for the variables in V.

Table 5.1: Table of symbols summarising the components of the action institution, $\mathcal{ACT}$ (Section 5.2.1), the UML state machine institution, $\mathcal{UML}$ (Section 5.2.3) and the institution for Event-B, $\mathcal{EVT}$ (Chapter 3).

SIGNATURES are tuples $\langle A_H, M_H, V_H \rangle$ where A_H is a set of action statements such as `x := x + 1`. M_H is a set of messages, which are typically signals that have been sent out by the actions; and V_H is a set of sort-indexed variables. Große-Rhode proposed a similar institution of actions, however, action labels were incorporated into the signature rather than action statements [57]. We discuss this approach in parallel with that taken by Knapp et al. [73] where relevant.

SENTENCES are expressions of the form

$$g_{pre} \rightarrow [a]\overline{m} \rhd g_{post}$$

where g_{pre} and g_{post} are sentences in the guard institution (predicates). These are similar to OCL constraints, in that if the precondition (g_{pre}) holds then after executing the action a, the messages $\overline{m}$ are output and the postcondition (g_{post}) holds.

MODELS describe transition relations over data states and consist of a set of 4-tuples of the form

$$\{\omega \xrightarrow{a,\overline{m}} \omega\prime\}$$

Here ω and $\omega\prime$ are data states of type $V_H \rightarrow Val$ and thus contain sort appropriate variable-to-value mappings for the variables in the V_H component of the action signature. As before, a is an action statement and $\overline{m}$ is the corresponding set of messages that are output. The approach taken by Große-Rhode uses action labels here rather than action statements and this provides more freedom when constructing sentences [57]. We have taken a similar approach in our institution for Event-B [40, 41].

SATISFACTION A model $\{\omega \xrightarrow{a,\overline{m}} \omega\prime\}$ satisfies a sentence $g_{pre} \rightarrow [a]\overline{m} \rhd g_{post}$ if and only if $\omega \models g_{pre} \wedge \omega\prime \models g_{post}$. This is very similar to the satisfaction condition evaluated in $\mathcal{EVT}$ (Definition 37).

We have outlined two different approaches to defining an action institution - that of Knapp et al. [73] using action statements and that of Große-Rhode [57] using action labels. We analyse each of these approaches below.

Action Statements vs Action Labels

There are positives and negatives to both of these approaches but we believe that action labels present a more liberal mechanism for describing action sentences. We view the action statement approach as a heavyweight way of ensuring that all details are captured explicitly, however, it seems counter-intuitive to include action sentences into the vocabulary component of the institution for actions. This is because sentences have to be defined over a specific signature and including sentences in the signature restricts us to writing specific action sentences (only those that are included in the signature) rather than deriving new sentences from the signature.

Of course, sentences in this institution comprise of actions and guards. It is possible to represent actions as before-after predicates as in the Event-B language [3]. From this point of view, actions and guards are both predicates. Guard statements may be built up from the signature whereas the set of action statements is predefined in the signature. The signature could be infinite so that all possible action statements are available for use. However, this is not very practical, particularly with regard to implementation. A more uniform approach is taken in $\mathcal{EVT}$ by viewing actions as before-after predicates. In fact, this approach also simplifies the evaluation of the satisfaction condition. In any case, the correspondence between the action institution and $\mathcal{FOPEQ}$ is visible, particularly when we embed $\mathcal{EVT}$-models into $\mathcal{FOPEQ}$ via comorphism in order to evaluate their satisfaction.

The action institution forms the basis of the institution for UML state machines and in Section 5.2.2, we describe how the institution for behavioural state machines is built over $\mathcal{ACT}$ [73].

5.2.2 THE BEHAVIOURAL STATE MACHINE INSTITUTION

The second component of the $\mathcal{UML}$ institution is the institution for behavioural state machines [73]. This institution is built over the action institution described in Section 5.2.1 and is composed of the following:

SIGNATURES are triples of the form $\Sigma = \langle E_\Sigma, F_\Sigma, S_\Sigma \rangle$ where E_Σ are (external) events, F_Σ are completion events[1] and S_Σ are states[2]. Morphisms in this category are injective functions.

SENTENCES are pairs $\langle s_0, T \rangle$ where $s_0 \in S_\Sigma$ is an initial state (for example, to the state labelled `Idle` in the state machine in Figure 5.3). T is a set of transitions from a state s to a state $s\prime$ written as:

$$s \xrightarrow[T]{p[g]/a,\bar{f}} s\prime \tag{5.1}$$

where $p \in E_\Sigma \cup F_\Sigma$ is a trigger event, g is a sentence in the guard institution, a is an action sentence and $\bar{f}$ is a set of completion events drawn from F_Σ. The intuition is that a transition from s to $s\prime$ can occur if the event p is triggered (provided that the guard g is true), and then an action, a, is executed and a set of completion events, $\bar{f}$ is output.

MODELS are pairs of the form $\Theta = \langle I_\Theta, \Delta_\Theta \rangle$ where

$$I_\Theta \in \wp(V_H \rightarrow Val) \times S_\Sigma$$

1 The completion events consist of those states from which a completion transition originates (states with no trigger events) [73].

2 Note that $E_\Sigma \cap F_\Sigma = \varnothing$.

represents initial configurations and

$$\Delta_\Theta \subseteq C_\Sigma \times \wp(M_H) \times C_\Sigma$$

where $C_\Sigma = (V_H \rightarrow Val) \times \wp(E_\Sigma \cup F_\Sigma) \times S_\Sigma$ represents a transition relation between configurations that emits messages. These messages correspond to signals that are output (such as method calls in the ATM example [73]). Each configuration is comprised of an action state, an event pool and a control state. Each element of Δ_Θ has the form

$$(\omega, p, s) \xrightarrow{\overline{m}} (\omega\prime, p\prime, s\prime) \in \Delta_\Theta$$

where $\omega \in V_H \rightarrow Val$ is a variable-to-value mapping (as is $\omega\prime$), p and $p\prime$ are triggering events, s and $s\prime$ are states and $\overline{m} \subseteq \wp(M_H)$ is a set of messages.

SATISFACTION the satisfaction relation $\langle I_\Theta, \Delta_\Theta \rangle \models_\Sigma \langle s_0, T \rangle$ holds if and only if $\pi_2(I_\Theta) = s_0$ and Δ_Θ is the least transition relation satisfying Equations 5.2 and 5.3 as follows:

$$\begin{aligned} &(\omega, p :: \overline{p}, s) \xrightarrow[\Delta_\Theta]{\overline{m} \setminus E_\Sigma} (\omega\prime, \overline{p} \lhd ((\overline{m} \cap E_\Sigma) \cup \overline{f}), s\prime) \\ &\qquad \mathit{if} \;\; \exists (s \xrightarrow[T]{p[g]/a, \overline{f}} s\prime) \quad \cdot \quad \omega \models g \wedge \omega \xrightarrow[\Omega]{a, \overline{m}} \omega\prime \end{aligned} \tag{5.2}$$

When some event p is drawn from the event pool $\overline{p}$, written as $p :: \overline{p}$, then the messages $\overline{m} \setminus E_\Sigma$ are emitted and both the accepted and completion events in $(\overline{m} \cap E_\Sigma) \cup \overline{f}$ are added to the event pool of the target configuration. Note that the set of messages is partitioned twice, first into those messages that are not (external) events and then into those that are either external or completion events.

$$\begin{aligned} &(\omega, p :: \overline{p}, s) \xrightarrow[\Delta_\Theta]{\varnothing} (\omega, \overline{p}, s) \\ &\qquad \mathit{if} \;\; \forall (s \xrightarrow[T]{p\prime[g]/a, \overline{f}} s\prime) \quad \cdot \quad p \neq p\prime \vee \omega \not\models g \end{aligned} \tag{5.3}$$

Here, when no transition is triggered by the current event, the event is discarded ($p :: \overline{p}$ becomes $\overline{p}$ along the transition described by Δ_Θ). We note that $\overline{m} \setminus E_\Sigma$ can be empty in the first case (no signals are sent out) but that in the second case it must be empty as no transition has been triggered by the event. Here, it is important to note that the data state, ω, and labelled state, s, have not changed across the transition. The only change is that the event p has not triggered anything so it has been dropped from the event pool. This corresponds in $\mathcal{EVT}$ to the event p whose guard is not satisfied by the data state ω, so no updates are made to the labelled state or to the data state as no actions occur.

5.2.3 THE PROTOCOL STATE MACHINE INSTITUTION

The third component of the $\mathcal{UML}$ institution is the institution for protocol state machines. These are similar to behavioural state machines but they use a precondition and postcondition to trigger a transition, instead of showing guards and effects. Protocol state machines inhabit a special designated error state when they encounter an event that does not fire a transition rather than discarding it, as was the approach in its behavioural counterpart (Section 5.2.2). In this scenario, signatures are the same as those described for behavioural state machines with a distinguished error state e included in the set of states S_Σ. Sentences are of the form (s_0, e, T) where

$$T \subseteq S_\Sigma \times (G(V_H) \times E_\Sigma \times G(V_H) \times \wp(M_H) \times \wp(F_\Sigma)) \times S_\Sigma$$

such that $G(V_H)$ denotes a guard sentence over the variables in V_H. Models are similar to those of behavioural state machines, except for in the second clause of Δ_Θ where the error state is targeted when no transition is enabled. In the behavioural state machine this would have resulted in the event being discarded as can be seen in Equation 5.3.

The satisfaction condition now maintains that when some event is chosen from the event pool, the precondition of some transition holds in the source configuration and its postcondition holds in the target configuration.

5.2.4 THE STATE MACHINE TRIPOD OF INSTITUTIONS

Knapp et al use a Grothendieck institution, as introduced in Section 2.4.2, to flatten these institutions [73]. Here, signatures are pairs $\langle H, \Sigma \rangle$ where H is a signature in the action institution, $\mathcal{ACT}$, and Σ is as described in Section 5.2.2. Sentences can either be sentences over the action institution or either behavioural or protocol state machine sentences (as described in Sections 5.2.2 and 5.2.3 respectively). Models are pairs $\langle \Omega, \Theta \rangle$ where Ω is a model over the action institution and Θ is as described above. The satisfaction condition is also inherited [73]. We refer the reader to Figure 5.2 for an illustration of the relationships between these institutions.

5.3 TRANSLATING FROM $\mathcal{UML}$ TO $\mathcal{EVT}$ VIA AN INSTITUTION COMORPHISM

In this section we define our institution comorphism from $\mathcal{UML}$ to $\mathcal{EVT}$ and prove that the comorphism satisfaction condition is preserved (Definition 214. We begin by discussing the similarities and the differences between $\mathcal{UML}$ and $\mathcal{EVT}$.

5.3.1 COMPARING $\mathcal{EVT}$ AND $\mathcal{UML}$

The $\mathcal{EVT}$ and $\mathcal{UML}$ institutions bear some similarities: they are both built over a primitive institution that provides a mathematical language

for use in their sentences. This is $\mathcal{FOPEQ}$ for $\mathcal{EVT}$ and the action institution, $\mathcal{ACT}$, for $\mathcal{UML}$. In fact, both rely on this primitive institution in order to evaluate their satisfaction conditions. With regard to the model component of $\mathcal{EVT}$ and $\mathcal{UML}$, both have a notion of transition states and both also include a representation of initial state in their models (the L component of an $\mathcal{EVT}$-model and the I_Θ component of an $\mathcal{UML}$-model).

However, there are points upon which they differ. The underlying action institution of $\mathcal{UML}$ includes messages such as method calls and is similar to dynamic logic [73]. $\mathcal{EVT}$ does not have these properties because events in Event-B cannot call methods or other events. $\mathcal{UML}$ distinguishes between completion and external events whereas $\mathcal{EVT}$ does not. A strength of $\mathcal{EVT}$ is that events can trigger non-deterministically as long as their guards evaluate to `true`. $\mathcal{EVT}$ has no explicit representation of state other than the data state which is represented using variable-to-value mappings. These data states are represented in the base (action) institution of $\mathcal{UML}$ but there are also labelled states in the signature that indicate the state names on a state machine diagram.

Our intent is to emulate the UML-B plugin which translates state machines constructed using a UML-like formalism into Event-B [115]. Therefore, we define an institution comorphism to translate sentences written in the language of the $\mathcal{UML}$ institution to sentences written in the language of the $\mathcal{EVT}$ institution.

The UML-B plugin uses an Event-B specific subset of UML that does not include messages or completion events. The interface is Event-B friendly in that it enables users to add parameters, guards, witnesses and actions to transitions. The user can select if a new event should be generated for a transition or if the transition should map to an event already defined in a machine in the same project. The plugin translates UML state diagrams into Event-B, but the user can decide how the states should be translated: either every state is a separate boolean flag or

there is one state variable whose type is defined in a context to have as values the names of the states in the state machine. Our choice is the latter, more elegant, approach in what follows.

5.3.2 RELATING THE ACTION INSTITUTION AND $\mathcal{FOPEQ}$

In order to achieve our goals we assume the existence of a comorphism between the underlying institutions of $\mathcal{UML}$ and $\mathcal{EVT}$ ($\mathcal{ACT}$ and $\mathcal{FOPEQ}$ respectively). We denote this assumed translation by

$$\rho_{BASE} : \mathcal{FOPEQ} \to \mathcal{ACT}$$

and outline the relationship here: the guard institution, over which the action institution is built, is $\mathcal{FOPEQ}$. We can write an action as a before-after predicate because the variable names are included in the action signature. For example, the action sentence

$$\mathtt{x} := \mathtt{x} + 1$$

becomes the before-after predicate

$$\overline{\mathtt{x}}\prime = \mathtt{x} + 1$$

Sentences in the guard institution are predicates so the above sentence is in fact a sentence in the guard institution.

When we defined a comorphism embedding from $\mathcal{EVT}$ to $\mathcal{FOPEQ}$ in Definition 38 we internalised the variables (both before and after ones) as operator symbols of the appropriate sort. We can use a similar construction to treat the action institution described by Knapp et al. as $\mathcal{FOPEQ}$. We will not present any more details on ρ_{BASE} here but we will refer to its existence in what follows.

5.3.3 RELATING THE $\mathcal{UML}$ INSTITUTION AND $\mathcal{EVT}$

We outline the comorphism relationship between $\mathcal{UML}$ and $\mathcal{EVT}$ in three parts. First we relate their signatures in Definition 51, then their sentences in Defintion 52 and their models in Definition 53. Finally, we prove that the comorphism we have defined preserves the axiomatic properties required of a comorphism (Theorem 51).

Definition 51. $\rho : \mathcal{UML} \to \mathcal{EVT}$ is a functor such that

$$\rho^{Sign}(\langle H, \Sigma \rangle) = \langle S, \Omega, \Pi, E, V \rangle$$

The translation of H follows ρ_{BASE} to give $\langle S, \Omega, \Pi \rangle$. Then, we translate $\Sigma = \langle E_\Sigma, F_\Sigma, S_\Sigma \rangle$ as follows:

- We take the union $E_\Sigma \cup F_\Sigma$ to form the set of (event name, status) pairs E. Each of the elements of E_Σ and F_Σ are paired with the status `ordinary`, as $\mathcal{UML}$ has no mechanism for evaluating or proving any convergence properties. The definition of this comorphism thus provides $\mathcal{UML}$ with a means for describing convergence properties.

- To represent the elements of S_Σ we construct the new sort *State* and add it to the set of sort names S in the $\mathcal{EVT}$ signature. The elements of S_Σ are added as 0-ary operators of sort *State*. We also form $V = V_H \cup \{state : State\}$ by including a single state variable to represent the current state.

In both $\mathcal{UML}$ and $\mathcal{EVT}$, signature morphisms are sort/arity-preserving renaming functions. The only differences in $\mathcal{EVT}$ are that the `Initialisation` event must be preserved by these renamings and there is a restriction as to how the status of events can be altered by signature morphisms i.e. they must preserve the poset ordering { `ordinary` $<$ `anticipated` $<$ `convergent` } (Definition 32).

As outlined in Definition 27, natural transformations relate functors and so we define a natural transformation between the **Sen** functors of the $\mathcal{UML}$ and $\mathcal{EVT}$ institutions in Definition 52.

Definition 52. $\rho^{Sen} : \rho^{Sign}; \mathbf{Sen}_{\mathcal{UML}} \rightarrow \mathbf{Sen}_{\mathcal{EVT}}$ is a natural transformation such that

$$\rho^{Sen}\langle s_0, T\rangle = \{\langle e, \phi(\overline{x}, \overline{x}\prime)\rangle\}$$

More specifically,

$$\rho^{Sen}(s_0, \{s \xrightarrow[T]{p[g]/a,\overline{f}} s\prime\}) = \{\langle p, \phi(\overline{x}, \overline{x}\prime)\rangle\}$$

based on the definition of T in Equation 5.1 and via the following:

- For every $\{s \xrightarrow{p[g]/a,\overline{f}} s\prime \in T\}$ the $\mathcal{EVT}$ event sentence $\langle p, \phi(\overline{x}, \overline{x}\prime)\rangle$ is formed where

  $$\phi(\overline{x}, \overline{x}\prime) = (\forall\, \overline{x}, \overline{x}\prime \cdot state = s \wedge \rho_{BASE}([g]/a) \wedge state\prime = s\prime)$$

  and the variables $\overline{x}$ and $\overline{x}\prime$ are drawn from V_H. Note that the event name corresponds to the trigger event p. The comorphism translation $\rho_{BASE}([g]/a)$ translates the guard and action components from $\mathcal{ACT}$ to $\mathcal{EVT}$. This means that we construct the formula $\phi(\overline{x}, \overline{x}\prime)$ which contains the guard and the action written as a before-after predicate. Completion events that appear on the transition, $\overline{f}$, are omitted as they play no role in $\mathcal{EVT}$.

- Since $\mathcal{EVT}$ does not require that events happen in a particular sequence, the ordering in the set of transitions T is implicitly forced using our additional state variable. This is similar to the approach taken by the UML-B plugin which does not use institution theory in its translation process [115].

- We form the sentence $\langle \texttt{Initialisation}, \phi(\overline{x}, \overline{x}\prime)\rangle$ where $\phi(\overline{x}, \overline{x}\prime) = (\forall\, \overline{x}, \overline{x}\prime \cdot state\prime = s_0)$ in order to ensure that the initial value of the *state* variable is set to the initial state in the $\mathcal{UML}$ state machine.

Note that we only produce events of status `ordinary` so there is no confusion over the status. The initial state in the $\mathcal{UML}$ behavioural state machine does not become an event, it is a variable update in the `Initialisation` event so any $\mathcal{UML}$-signature morphism that is applied to s_0 does not adversely affect the preservation of the `Initialisation` event.

Dealing with Multiple Method Calls

We identified two distinct approaches to the definition of this natural transformation, both arise as a result of multiple method calls appearing on a $\mathcal{UML}$-state transition. As events in Event-B share some similarity with the concept of methods in a programming paradigm, the first approach is to translate method calls that appear on $\mathcal{UML}$-state transitions into individual event sentences in $\mathcal{EVT}$. However, if a transition contains multiple method calls then multiple events are generated (one for each method call) in the corresponding $\mathcal{EVT}$-specification. In Event-B, these events are triggered in any order once their guards hold. Therefore, in one trace of the Event-B machine, it is possible that not all of these events are triggered while moving from their source to their target state. Essentially, this approach offers a choice between these methods whereas their appearance on the $\mathcal{UML}$-state transition ensures that they all occur when moving between the source and target states. A possible solution is to add boolean flags during this translation. Although burdensome, these can be used to ensure that all method calls happen in order to progress from the source to target state.

The second approach involves representing each $\mathcal{UML}$-state transition as an individual event in $\mathcal{EVT}$. In this setting, the problem of multiple method calls can be mediated by introducing intermediate states in the state machine using a preprocessing phase before the sentence translation component of this comorphism takes place. Intermediate states

are only required when there is more than one method call appearing on the transition and the total number of intermediate states required is one less than the number of method calls that appear on the transition. One caveat of this approach is that if two transitions execute the same event, they are distinguished in the preprocessing phase. In order to ensure the correct evaluation of preconditions and postconditions the first transition will contain the guard and the last transition will contain the postcondition/action(s) that is to be reached.

While both of these approaches are viable, we opted for the second one as it is a more elegant solution. In either case, once the translation has occurred there is no way to distinguish between events that are generated and those that are consumed along a transition. In $\mathcal{EVT}$, however, we do not view this distinction as a crucial requirement of the translation. With regard to method calls that take parameters, these parameters are included as global variables in the corresponding $\mathcal{EVT}$-specification. We introduce event parameters that are constrained (in the guard part of the corresponding event) to have the value of these global variables. This is an important feature of the sentence translation so that, if a shared variable/event approach to modularisation is used at a later stage in Event-B, the event will be partitioned correctly. Alternatively, we could add an action statement to indicate that the values of these variables do not change during the event. Our approach using parameters is more flexible, particularly in the setting of a refinement step in $\mathcal{EVT}$/Event-B.

Next, we define a natural transformation between the **Mod** functors of the $\mathcal{UML}$ and $\mathcal{EVT}$ institutions in Definition 53.

Definition 53. $\rho^{Mod} : (\rho^{Sign})^{op};\ \mathbf{Mod}_{\mathcal{EVT}} \to \mathbf{Mod}_{\mathcal{UML}}$ is a natural transformation such that

$$\rho^{Mod}(\langle A, L, R\rangle) = \langle \Omega, \Theta \rangle$$

where Ω is a model over $\mathcal{ACT}$ and $\Theta = \langle I_\Theta, \Delta_\Theta \rangle$ is a behavioural/protocol state machine institution model.

$$\rho_{BASE}(A) = \Omega$$

Then,

$$\rho^{Mod}(\langle L, R \rangle) = \Theta = \langle I_\Theta, \Delta_\Theta \rangle$$

is achieved by the following:

- L is transformed into I_Θ by pairing L with the value of the `state'` variable as assigned by the `Initialisation` event. Hence, $I_\Theta = \langle \{state'\} \lessdot L, \{state'\} \lhd L \rangle$ where $\lessdot$ and $\lhd$ denote domain anti-restriction and domain restriction respectively.

- Each element of Δ_Θ is of the form

  $$(\omega, p, s) \xrightarrow{\overline{m}} (\omega', p', s')$$

  Each element of $R_p \in R$ is indexed by its event name p and is of the form:

  $$(\omega \cup \{state \mapsto s\}, \omega' \cup \{state' \mapsto s'\})_p$$

  Δ_Θ is constructed by extracting the index event as the triggering event p in the above formula, and the value of the designated *state* and *state'* variables to form the transition relation:

  $$(\omega, p, s) \xrightarrow{\varnothing} (\omega', p, s')$$

  Note that the value of the triggering event p is the same on both sides of this equation and no messages are output. We could have modelled the behaviour of the stream of messages using a variable in $\mathcal{EVT}$ but we viewed this as an unnecessary addition that would complicate the definition of this comorphism so it is omitted from further discussion.

Definitions 51, 52 and 53 combine to define the institution comorphism $\rho : \mathcal{UML} \to \mathcal{EVT}$ such that for any signature $\Sigma \in |\mathbf{Sign}_{\mathcal{UML}}|$, the translations $\rho^{Sen}_{\Sigma} : \mathbf{Sen}_{\mathcal{UML}}(\Sigma) \to \mathbf{Sen}_{\mathcal{EVT}}(\rho^{Sign}(\Sigma))$ of sentences and $\rho^{Mod}_{\Sigma} : \mathbf{Mod}_{\mathcal{UML}}(\rho^{Sign}(\Sigma)) \to \mathbf{Mod}_{\mathcal{EVT}}(\Sigma)$ of models preserve the satisfaction relation, that is, for any $\psi \in \mathbf{Sen}_{\mathcal{UML}}(\Sigma)$ and $M' \in |\mathbf{Mod}_{\mathcal{UML}}(\rho^{Sign}(\Sigma))|$

$$\rho^{Mod}_{\Sigma}(M') \models^{\mathcal{UML}}_{\Sigma} \psi \quad \Leftrightarrow \quad M' \models^{\mathcal{EVT}}_{\rho^{Sign}(\Sigma)} \rho^{Sen}_{\Sigma}(\psi) \tag{5.4}$$

We now prove that $\rho : \mathcal{UML} \to \mathcal{EVT}$ is a valid institution comorphism.

Theorem 51. $\rho : \mathcal{UML} \to \mathcal{EVT}$ *is a valid institution comorphism.*

Proof. Our objective is to prove that for any signature $\Sigma \in |\mathbf{Sign}_{\mathcal{UML}}|$, the translations $\rho^{Sen}_{\Sigma} : \mathbf{Sen}_{\mathcal{UML}}(\Sigma) \to \mathbf{Sen}_{\mathcal{EVT}}(\rho^{Sign}(\Sigma))$ of sentences and $\rho^{Mod}_{\Sigma} : \mathbf{Mod}_{\mathcal{UML}}(\rho^{Sign}(\Sigma)) \to \mathbf{Mod}_{\mathcal{EVT}}(\Sigma)$ of models preserve the satisfaction relation, that is, for any $\psi \in \mathbf{Sen}_{\mathcal{UML}}(\Sigma)$ and $M' \in |\mathbf{Mod}_{\mathcal{UML}}(\rho^{Sign}(\Sigma))|$

$$\rho^{Mod}_{\Sigma}(M') \models^{\mathcal{UML}}_{\Sigma} \psi \quad \Leftrightarrow \quad M' \models^{\mathcal{EVT}}_{\rho^{Sign}(\Sigma)} \rho^{Sen}_{\Sigma}(\psi)$$

Our proof is completed by showing that the satisfaction relations, $\models_{\mathcal{UML}}$ and $models_{\mathcal{EVT}}$, on each side of the $\Leftrightarrow$ above are equivalent. We begin by making the following substitutions based on the definitions provided earlier. Let $M' = \langle A, L, R \rangle$, $\psi = \langle s_0, T \rangle$ and $\Sigma = \langle H, \Sigma \rangle$. We must show:

$$\rho^{Mod}_{\langle H,\Sigma \rangle}(\langle A, L, R \rangle) \models^{\mathcal{UML}}_{\langle H,\Sigma \rangle} \langle s_0, T \rangle \quad \Leftrightarrow \quad \langle A, L, R \rangle \models^{\mathcal{EVT}}_{\rho^{Sign}(\langle H,\Sigma \rangle)} \rho^{Sen}_{\langle H,\Sigma \rangle}(\langle s_0, T \rangle) \tag{5.5}$$

From Definition 51, we know that $\rho^{Sign}(\langle H, \Sigma \rangle) = \langle S, \Omega, \Pi, E, V \rangle$. This is a simple translation and so we omit it from further discussion in order to simplify the notation. We also omit the subscript on ρ where obvious as we have done in the above definition, this gives

$$\rho^{Mod}(\langle A, L, R \rangle) \models^{\mathcal{UML}} \langle s_0, T \rangle \quad \Leftrightarrow \quad \langle A, L, R \rangle \models^{\mathcal{EVT}} \rho^{Sen}(\langle s_0, T \rangle)$$

From Definition 52, $\rho^{Sen}(\langle s_0, T\rangle)$ yields the set of $\mathcal{EVT}$-sentences

$$\rho^{Sen}(\langle s_0, T\rangle) = \left\{ \begin{array}{l} \langle \texttt{Initialisation}, \forall \overline{x}, \overline{x}\prime \cdot state\prime = s_0\rangle, \\ \{\langle p, \forall \overline{x}, \overline{x}\prime \cdot state = s \wedge \rho_{BASE}(g \wedge a) \wedge state\prime = s\prime\rangle\} \end{array} \right\}$$

Thus The satisfaction relation to be evaluated on the right hand side becomes

$$\langle A, L, R\rangle \models^{\mathcal{EVT}} \left\{ \begin{array}{l} \langle \texttt{Initialisation}, \forall \overline{x}, \overline{x}\prime \cdot state\prime = s_0\rangle, \\ \{\langle p, \forall \overline{x}, \overline{x}\prime \cdot state = s \wedge \rho_{BASE}(g \wedge a) \wedge state\prime = s\prime\rangle\} \end{array} \right\}$$

which is reduced to satisfaction over $\mathcal{FOPEQ}$ in the usual way for $\mathcal{EVT}$ (embedding the $\mathcal{EVT}$-model into $\mathcal{FOPEQ}$ using the comorphism described in Chapter 3).

Then, on the left hand side, $\rho^{Mod}(\langle A, L, R\rangle) = \langle \Omega, \Theta\rangle$ where $\Theta = \langle I_\Theta, \Delta_\Theta\rangle$ and I_Θ is formed by pairing the variable-to-value mappings in L (without a reference to the state variable $state\prime$) with the initial state obtained using the domain anti-restriction $\{state\prime\} \lhd L$. This part of the satisfaction condition corresponds to a simple equality check that $L(state\prime) = s_0$. This is always true as it is the value of the initial state. Moreover, since the satisfaction of the Δ_Θ component of the model is evaluated over $\mathcal{ACT}$, and we have assumed the existence of a mapping between it and $\mathcal{FOPEQ}$, then the satisfaction conditions on either side of the $\Leftrightarrow$ above are equivalent and so the comorphism property holds.

$\square$

In this section, we defined the comorphism relationship between $\mathcal{UML}$ and $\mathcal{EVT}$. In order to illustrate this comorphism in practice we apply it to the ATM state machine in Figure 5.3 in Section 5.4.

5.4 EXAMPLE

In order to illustrate the comorphism in practice we apply it to the ATM state machine used by Knapp et al. [73] and shown in Figure 5.3. This

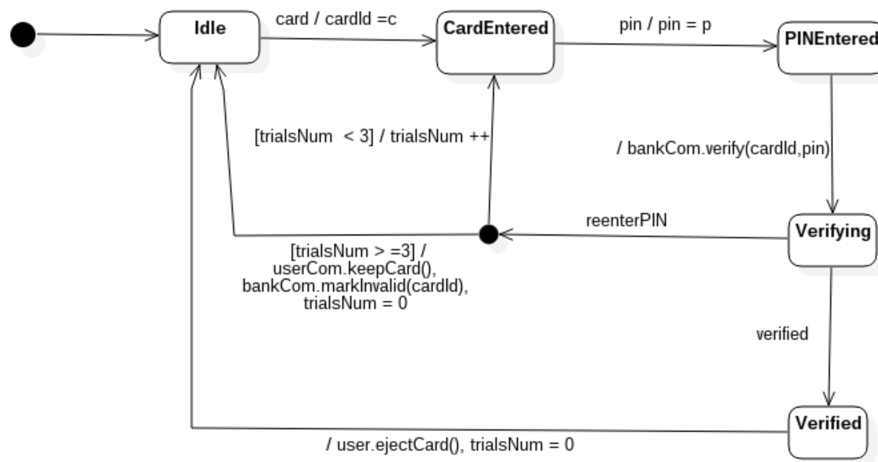


Figure 5.3: UML state machine describing an automated teller machine (ATM) based on the example in Knapp et al. [73].

machine has five states, representing a user entering an ATM card and a PIN number, which is then verified. The transitions between the states are labelled with events, guards and actions as described in Section 5.2.2. For example, from Figure 5.3 we can see that there is a transition from state `CardEntered` to the state `PINEntered` triggered by the event `pin` and which performs the action `pin = p`.

Preprocessing

We preprocess the state machine pesented in Figure 5.3 to produce the state machine shown in Figure 5.4. This involves the introduction of the new intermediate state `Inter1` in order to sequence the method calls on the original `Verifying` to `Idle` transition in Figure 5.3. We illustrate this preprocessed state machine in Figure 5.4.

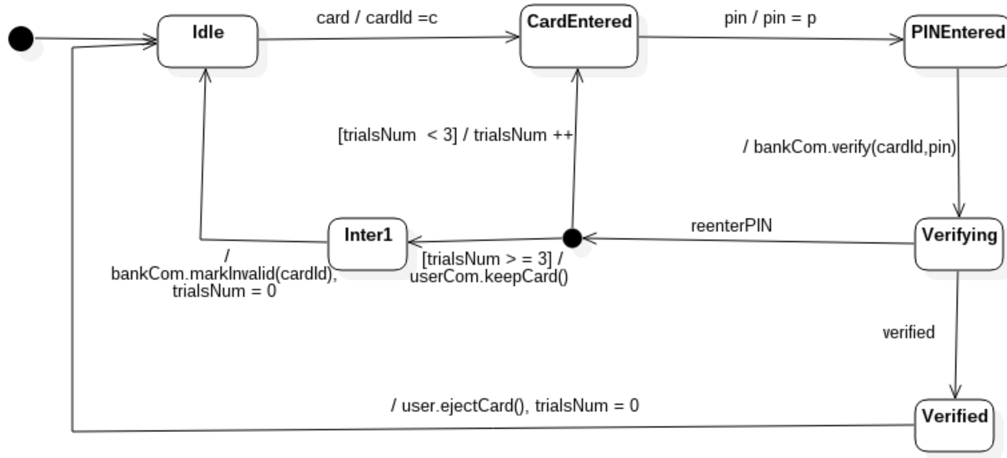


Figure 5.4: Preprocessed version of the state machine from Figure 5.3. Note that we have added the intermediate state `Inter1` during the preprocessing phase in order to split up the method calls `userCom.keepCard()` and `bankCom.markInvalid(cardId)`.

Signature Extraction and Translation

Given the behavioural state machine of the ATM as described in Figure 5.4, we extract the $\mathcal{UML}$-signature:

$$\Sigma_{ATM} = \langle E_{ATM}, F_{ATM}, S_{ATM} \rangle$$

where

$$E_{ATM} = \{\texttt{card, pin, reenterPIN, verified}\}$$
$$F_{ATM} = \{\texttt{PINEntered, Verified, Inter1}\}$$
$$S_{ATM} = \{\texttt{Idle, CardEntered, PINEntered, Verifying, Verified, Inter1}\}$$

We note that `PINEntered` occurs both as a state and a completion event [73]. This is because the transition between `PINEntered` and `Verifying` has no explicit trigger declared. In this case, the machine creates a

trigger event with the same name as the state it is leaving in accordance with the UML 2.5 specification document Section 14.2.3.8.3 [1].

Then, $\rho^{Sign}(\Sigma_{ATM}) = \langle S, \Omega, \Pi, E, V \rangle$ where

$$State \in S$$

$$\{\texttt{Idle}, \texttt{CardEntered}, \texttt{PINEntered}, \texttt{Verifying}, \texttt{Verified}, \texttt{Inter1}\} \subseteq \Omega$$

$$\begin{aligned} E =& \{(\texttt{card}, \texttt{ordinary}), (\texttt{pin}, \texttt{ordinary}), (\texttt{reenterPIN}, \texttt{ordinary}),\\ & (\texttt{verified}, \texttt{ordinary}), (\texttt{PINEntered}, \texttt{ordinary}),\\ & (\texttt{Verified}, \texttt{ordinary}), (\texttt{Inter1}, \texttt{ordinary})\\ & ,(\texttt{Initialisation}, \texttt{ordinary})\}\\ V =& V_H \cup \{\texttt{state:State}\} \end{aligned}$$

Once the signature has been extracted, we can extract the $\mathcal{UML}$-sentences and apply the translation ρ^{Sen} to them.

Sentence Extraction and Translation

We extract the following $\mathcal{UML}$-sentence $\langle s_0, T \rangle$ from Figure 5.4 where

$$\begin{aligned} s_0 =& \texttt{Idle}\\ T =& \{\texttt{Idle} \xrightarrow[T]{\texttt{card[true]/cardId = c,}\varnothing} \texttt{CardEntered},\\ & \texttt{CardEntered} \xrightarrow[T]{\texttt{pin[true]/pin = p , PINEntered}} \texttt{PINEntered},\\ & \texttt{PINEntered} \xrightarrow[T]{\texttt{PINEntered[true]/bank.verify(cardId, pin),}\varnothing} \texttt{Verifying},\\ & \texttt{Verifying} \xrightarrow[T]{\texttt{reenterPIN[trialsNum<3]/trialsNum++,}\varnothing} \texttt{CardEntered},\\ & \ldots\} \end{aligned}$$

Then $\rho^{Sen}(\langle s_0, T\rangle)$ yields the following set of $\mathcal{EVT}$-sentences:

$$\left\{\begin{array}{l}
\left\langle \quad \texttt{Initialisation}, \forall \overline{x\prime} \cdot state\prime = \texttt{Idle} \right\rangle, \\
\left\langle \begin{array}{l} \texttt{card}, \forall \overline{x}, \overline{x\prime} \cdot state = \texttt{Idle} \wedge \rho_{BASE}(\texttt{[true]/cardId = c}) \\ \quad \wedge\, state\prime = \texttt{CardEntered} \end{array} \right\rangle, \\
\left\langle \begin{array}{l} \texttt{pin}, \forall \overline{x}, \overline{x\prime} \cdot state = \texttt{CardEntered} \;\; \wedge \rho_{BASE}(\texttt{[true]/pin = p}) \\ \quad \wedge\, state\prime = \texttt{PINEntered} \end{array} \right\rangle, \\
\left\langle \begin{array}{l} \texttt{PINEntered}, \forall \overline{x}, \overline{x\prime} \cdot state = \texttt{PINEntered} \\ \quad \wedge\, \rho_{BASE}(\texttt{[true]/bank.verify(cardId, pin)}) \\ \quad \wedge\, state\prime = \texttt{Verifying} \end{array} \right\rangle, \\
\left\langle \begin{array}{l} \texttt{reenterPIN}, \forall \overline{x}, \overline{x\prime} \cdot state = \texttt{Verifying} \\ \quad \wedge\, \rho_{BASE}(\texttt{[trialsNum<3]/trialsNum++}) \wedge state\prime = \texttt{Verified} \end{array} \right\rangle, \\
\ldots
\end{array}\right\}$$

In this section we described how the comorphism translation applies to the signature and sentence components of a $\mathcal{UML}$-specification. In the next section we illustrate the preservation of the satisfaction condition (Theorem 51) for the comorphism in light of the ATM example presented in Figure 5.3.

5.4.1 PRESERVATION OF THE SATISFACTION CONDITION

In this section, we illustrate the proof of Theorem 51 in light of the ATM example. This section will be of particular interest to readers that are not familiar with institution theory as a means for conveying the veracity of our proof. Recall that an $\mathcal{EVT}$-model of the $\mathcal{EVT}$-signature that satisfies the $\mathcal{EVT}$-sentences obtained above is of the form $\langle A, L, R\rangle$ (Definition 35) where A is a $\mathcal{FOPEQ}$-model, L is the initialising set and R is a set containing event-indexed variable-to-value mappings for the be-

fore and after states of every $\mathcal{EVT}$-sentence (except those containing the `Initialisation` event) described above. Thus the satisfaction relation

$$\langle A, L, R \rangle \models_{\mathcal{EVT}} \rho^{Sen}(\langle s_o, T \rangle)$$

is evaluated for every sentence in $\rho^{Sen}(\langle s_0, T \rangle)$ using the embedding that we described in Section 3.5 as follows:

$$\begin{aligned}
\langle A, L, R \rangle \models_{\mathcal{EVT}} & \langle \texttt{Init}, \forall \overline{x}\prime \cdot state\prime = \texttt{Idle} \rangle \\
& \Leftrightarrow \forall s\prime \in L \cdot A^{(s\prime)} \models_{\mathcal{FOPEQ}} \forall \overline{x}\prime \cdot state\prime = \texttt{Idle} \\
\langle A, L, R \rangle \models_{\mathcal{EVT}} & \langle \texttt{card}, \forall \overline{x}, \overline{x}\prime \cdot state = \texttt{Idle} \\
& \quad \wedge \rho_{BASE}(\texttt{[true]/cardId = c}) \wedge state\prime = \texttt{CardEntered} \rangle \\
& \Leftrightarrow \forall \langle s, s\prime \rangle \in R_{\texttt{card}} \cdot A^{(s,s\prime)} \models_{\mathcal{FOPEQ}} state = \texttt{Idle} \\
& \quad \wedge \rho_{BASE}(\texttt{[true]/cardId = c}) \wedge state\prime = \texttt{CardEntered} \\
& \ldots
\end{aligned}$$

Then, $\rho^{Mod}(\langle A, L, R \rangle)$ gives the $\mathcal{UML}$-model

$$\langle \Omega, \langle I_\Theta, \Delta_\Theta \rangle \rangle$$

where Ω is a model in the action institution that we can embed into $\mathcal{FOPEQ}$ in a similar way to $A^{(s\prime)}$ and $A^{(s,s\prime)}$ as outlined in Section 5.3.2. In order to construct I_Θ we use domain restriction $\{state\prime\} \lhd L$ to obtain the value of the $state\prime$ variable. We remove all mappings that correspond to the $state\prime$ variable as it does not appear in the $\mathcal{UML}$ state machine using domain anti-restriction ($⩤$). Then

$$I_\Theta = (\{state\prime\} ⩤ L, \{state\prime\} \lhd L)$$

The set of transition relations Δ_Θ formed from R contains

$$(\omega, p, s) \xrightarrow{\varnothing} (\omega\prime, p, s\prime)$$

for every element of each $R_p \in R$ where p is an event name. Here s and $s\prime$ are obtained using the projection of the $state$ and $state\prime$ variable values,

and these mappings are then omitted from the data states denoted by ω and $\omega\prime$ in the same way as for the construction of I_Θ above.

Evaluating the satisfaction condition

$$\rho^{Mod}(\langle A, L, R\rangle) \models_{\mathcal{UML}} (s_o, T)$$

involves ensuring that $\pi_2(I_\Theta) = s_0$. Using the translation above, this simplifies to `Idle` $=$ `Idle` which evaluates to `true`. Then we ensure that the following holds for each of the transition relations in Δ_Θ outlined above

$$\exists\, s \xrightarrow{p[g]/a,\bar{f}} s\prime \in T \quad \cdot \quad \omega \models g \wedge (\omega \xrightarrow{a\overline{m}} \omega\prime \in \Omega)$$

This can be further simplified, by embedding Ω into $\mathcal{FOPEQ}$, to

$$\exists\, s \xrightarrow{p[g]/a,\bar{f}} s\prime \in T \quad \cdot \quad \Omega \models_{\mathcal{FOPEQ}} \forall(x, \overline{x}\prime) \in (\omega \cup \omega\prime) \cdot \rho_{BASE}(g \wedge a)$$

which is equivalent to the $\mathcal{EVT}$-satisfaction condition described in Section 3.5. This equivalence was core to the proof of Theorem 51 and we have shown that our example behaves as expected.

5.4.2 ANALYSING A SELECTION OF POTENTIAL EDGE CASES

We consider the potential edge cases of the proof of Theorem 51 and argue that they do not affect our construction of the comorphism presented in Section 5.3.

1. If the guard or action is a contradiction then the satisfaction condition holds because both satisfaction conditions are evaluated in the base institution of the relevant formalism. We have shown that these conditions are equivalent above.

2. There will always be variables present in V because the state variables will always be added to the signature. Therefore, we do not need to consider this case further.

3. A state machine will always have states as there must always be at least the start state, here the set S_Σ can never be empty.

4. If there are no (trigger) events in the $\mathcal{UML}$-signature (E_Σ) then all of the transitions are completion transitions. The events in $\mathcal{EVT}$ will thus be the completion events (which are states) in F_Σ.

5. If a state has multiple outgoing transitions with no triggers multiple events of the state's name could be produced. In this case these events are merged, even if they have contradicting guards. We address this problem by generating a new event name each time.

In this section, we used the illustrative example of an ATM state machine as described by Knapp et al. to show how our comorphism works in practice [73]. This corresponds to the institution comorphism translation indicated on the right of Figure 5.1. Next, we show how this translation provides a mathematical grounding for the UML-B Rodin plugin and our EB2EVT tool. This corresponds to the remainder of Figure 5.1.

5.5 THE COMORPHISM AS A FOUNDATION FOR UML-B

In this section, we illustrate how the comorphism that we have defined provides a mathematically sound basis for the approach to interoperability achieved by the UML-B Rodin plugin. Figure 5.5 specifies the ATM state machine using the UML-B plugin [114]. The labelled transition coming from the start state corresponds to the `Initialisation` event as indicated in Figure 5.5. As documented in the UML documentation Section 14.2.3.7 [1], the initial state can be the source of at most one transition. In contrast, the UML-B plugin allows modelling of more than one transition from the initial state. When the corresponding Event-B specification is generated an error occurs because the state

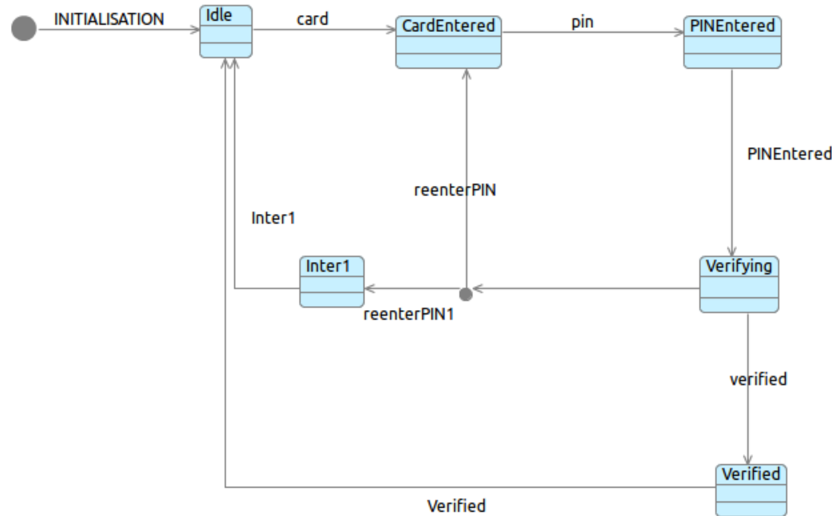


Figure 5.5: UML-B representation of the ATM state machine described in Figure 5.4. Note the addition of the completion events `PINEntered`, `Inter1` and `Verified` as transitions.

variable updates are given the same label. This is a small shortcoming of this plugin. Due to limitations with the plugin, the junction state caused some issues, and we had to add the transition `reenterPIN1` so that there were no issues with the translation to Event-B.

The Event-B specification generated by the UML-B model described in Figure 5.5 is shown in Figures 5.7 and 5.8. The context in Figure 5.7 provides for the inclusion of UML states as a new data type. This data type, called *behavioural_STATES* (line 2) has its carrier set partitioned via an axiom (line 11). The corresponding ATM machine (Figure 5.5) uses a variable called *behavioural* to keep track of the states before and after each event. This imposes control on the Event-B model. The UML-B translation also includes intuitive labels for the Event-B elements (guards, actions etc.) as seen in Figure 5.8.

We formalised a translational semantics of Event-B in Chapter 4 that allows the translation of specifications written in Event-B into specifications written over the institution $\mathcal{EVT}$. We concentrate here on an analysis of the raw signature and sentence output in order to demon-

strate our comorphism as described by Definitions 51 – 53 and Theorem 51.

We first extract the $\mathcal{EVT}$ signature from the specification in Figures 5.7 and 5.8 using our EB2EVT tool. This gives the signature $\Sigma = \langle S, \Omega, \Pi, E, V \rangle$ with

$$
\begin{aligned}
S &= \{behavioural_STATES\}\\
\Omega &= \{(Idle, behavioural_STATES), (CardEntered, behavioural_STATES),\\
&\quad (PINEntered, behavioural_STATES), (Verifying, behavioural_STATES),\\
&\quad (Verified, behavioural_STATES), (Inter1, behavioural_STATES)\}\\
\Pi &= \{\}\\
E &= \{(\texttt{card,ordinary}), (\texttt{pin,ordinary}), (\texttt{PINEntered,ordinary}),\\
&\quad (\texttt{verified,ordinary}), (\texttt{reenterPIN,ordinary}),\\
&\quad (\texttt{userCom.keepCard,ordinary}), (\texttt{bankCom.markInvalid, ordinary}),\\
&\quad (\texttt{Verified, ordinary}), (\texttt{Inter1,ordinary})\}\\
V &= \{(pin, \mathbb{N}), (cardId, \mathbb{N}), (trialsNum, \mathbb{N}),\\
&\quad (behavioural, behavioural_STATES)\}
\end{aligned}
$$

This signature corresponds to the signature generated by the application of ρ^{Sign} to the signature of the ATM machine over the $\mathcal{UML}$ institution. Here, the state variable is given by the variable *behavioural* and the state sort is given by the *behavioural_STATES* data type as generated by the UML-B plugin. The elements of the carrier set for *behavioural_STATES* are included here as 0-ary operators.

The $\mathcal{EVT}$-sentences shown in Figure 5.6 are then generated. Sentence 1 is generated by applying the comorphism from $\mathcal{FOPEQ}$ to $\mathcal{EVT}$ to the context shown in Figure 5.7. Each of the remaining sentences (lines 2–10) are generated for a particular event using the translational semantics that we defined in Section 4.6. These sentences correspond to those generated by the application of ρ^{Sen} to the ATM sentences over $\mathcal{UML}$ described earlier.

```
1  ⟨Init, partition(behavioural_STATES, {Idle}, {CardEntered}, {PINEntered}, {Verifying}, {Verified}, {Inter1})⟩
2  ⟨Init, (∀ x̄, x̄′ · pin′ :∈ ℕ ∧ cardId′ :∈ ℕ ∧ trialsNum′ = 0 ∧ behavioural′ = Idle)⟩
3  ⟨card, (∀ x̄, x̄′, c : ℕ · behavioural = Idle ∧ behavioural′ = CardEntered ∧ cardId′ = c)⟩
4  ⟨pin, (∀ x̄, x̄′, p : ℕ · behavioural = CardEntered ∧ behavioural′ = PINEntered ∧ pin′ = p)⟩
5  ⟨PINEntered, (∀ x̄, x̄, c, p : ℕ′ · behavioural = PINEntered ∧ c = cardId, p = pin ∧
   behavioural′ = Verifying)⟩
6  ⟨verified, (∀ x̄, x̄′ · behavioural = Verifying
   ∧ behavioural′ = Verified)⟩
7  ⟨Verified, (∀ x̄, x̄′ · behavioural = Verified ∧ behavioural′ = Idle ∧ trialsNum′ = 0)⟩
8  ⟨reenterPIN, (∀ x̄, x̄′ · behavioural = Verifying ∧ trialsNum < 3 ∧
   behavioural′ = CardEntered ∧ trialsNum′ = trialsNum + 1)⟩
9  ⟨Inter1, (∀ x̄, x̄′, c : ℕ · c = cardId ∧ behavioural = Inter1 ∧ trialsNum′ = 0 ∧
   behavioural′ = Idle)⟩
10 ⟨reenterPIN1, (∀ x̄, x̄′ · trialsNum ⩾ 3 ∧ behavioural = Verifying ∧ behavioural′ = Inter1)⟩
```

Figure 5.6: These are the $\mathcal{EVT}$-sentences extracted from the Event-B specification shown in Figure 5.8. $\bar{x}$ and $\bar{x}'$ can be inferred from V, described above.

```
1  CONTEXT behaviour_implicitContext
2    SETS behavioural_STATES
3    CONSTANTS Idle, CardEntered, PINEntered, Verifying, Verified, Inter1
4    AXIOMS
5        typeof_Idle: Idle ∈ behavioural_STATES
6        typeof_CardEntered: CardEntered ∈ behavioural_STATES
7        typeof_PINEntered: PINEntered ∈ behavioural_STATES
8        typeof_Verifying: Verifying ∈ behavioural_STATES
9        typeof_Verified: Verified ∈ behavioural_STATES
10       typeof_Inter1: Inter1 ∈ behavioural_STATES
11       distinct_states_in_behavioural_STATES: partition(behavioural_STATES, {Idle}, {CardEntered},
   {PINEntered}, {Verifying}, {Verified}, {Inter1})
12 END
```

Figure 5.7: This is the context that was generated by the UML-B plugin. It specifies a new datatype called *behavioural_STATES* that allows us to refer to the states in the corresponding UML state machine.

```
1  MACHINE behaviour   SEES behaviour_implicitContext
2    VARIABLES pin, cardId, trialsNum, behavioural
3    INVARIANTS  pin_type: pin ∈ ℕ
4                cardId_type: cardId ∈ ℕ
5                trialsNum_type: trialsNum ∈ ℕ
6                typeof_behavioural: behavioural ∈ behavioural_STATES
7    EVENTS
8    Initialisation ordinary
9      then  pin_init: pin :∈ ℕ
10           cardId_init: cardId :∈ ℕ
11           trialsNum_init: trialsNum := 0
12           init_behavioural: behavioural := Idle
13   Event card ≙ordinary
14     any c
15     when  c_type: c ∈ ℕ
16          isin_Idle: behavioural = Idle
17     then  enter_CardEntered: behavioural := CardEntered
18          card_Action1: cardId := c
19   Event pin ≙ordinary
20     any p
21     when  p_type: p ∈ ℕ
22          isin_CardEntered: behavioural = CardEntered
23     then  enter_PINEntered: behavioural := PINEntered
24          PIN_Action1: pin := p
25   Event PINEntered ≙ordinary
26     any c,p
27     when  c_type: c ∈ ℕ
28          p_type: p ∈ ℕ
29          behavioural_guards1: c = cardId
30          behavioural_guards2: p = pin
31          isin_PINEntered: behavioural = PINEntered
32     then  enter_Verifying: behavioural := Verifying
33   Event verified ≙ordinary
34     when  isin_Verifying: behavioural = Verifying
35     then  enter_Verified: behavioural := Verified
36   Event Verified ≙ordinary
37     when  isin_Verified: behavioural = Verified
38     then  behavioural_actions3: trialsNum := 0
39          enter_Idle: behavioural := Idle
40   Event reenterPIN ≙ordinary
41     when  behavioural_guards6: trialsNum < 3
42          isin_Verifying: (behavioural = Verifying)
43     then  behavioural_actions6: trialsNum := trialsNum + 1
44          enter_CardEntered: behavioural := CardEntered
45   Event Inter1 ≙ordinary
46     any c
47     when  c_type: c ∈ ℕ
48          behavioural_guards5: c = cardId
49          isin_Inter1: behavioural = Inter1
50     then  behavioural_actions5: trialsNum := 0
51          enter_Idle: behavioural := Idle
52   Event reenterPIN1 ≙ordinary
53     when  behavioural_guards7: trialsNum ⩾ 3
54          isin_Verifying: (behavioural = Verifying)
55     then  enter_Inter1: behavioural := Inter1
56 END
```

Figure 5.8: The Event-B machine generated from the UML-B model shown in Figure 5.5. Notice that every transition in the state machine corresponds to an event in this Event-B machine.

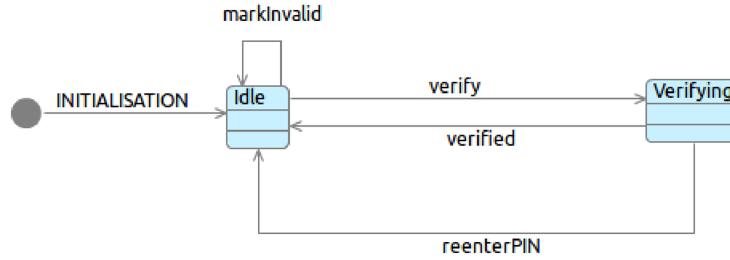


Figure 5.9: Protocol state machine from [73] as represented using the UML-B plugin.

5.5.1 REFINEMENT

As described in Sections 2.1.1 and 2.3.4, refinement is a feature of both the Event-B formal specification language and the theory of institutions. By extension therefore, UML-B and the institutions for $\mathcal{UML}$ and $\mathcal{EVT}$ all obey some basic refinement properties. Knapp et al. use an ATM protocol state machine to illustrate the refinement relationship between state machines [73]. We present this abstract state machine using UML-B in Figure 5.9. The machine in Figure 5.3 is a refinement of this protocol state machine. Refinement outlined by Knapp et al. simply involves a check for trace inclusion [73]. More formally, this means that for an abstract state machine SM_a and a concrete state machine SM_c, the refinement, $SM_a \sqsubseteq SM_c$ holds if for all Σ-models M,

$$M\mid_{\sigma} \in Mod(SM_c) \quad \Rightarrow \quad M\mid_{\theta} Mod(SM_a)$$

Knapp et al. have shown that this is true for some "mediating signature" Σ and morphisms $\theta : Sig(SM_a) \to \Sigma$ and $\sigma : Sig(SM_c) \to \Sigma$. They use the notion of theory translation along a signature morphism to show this.

We can generate an abstract Event-B specification corresponding to the state machine of Figure 5.9, and this is shown in Figure 5.11. In UML-B, it is possible to refine state machines. However, context ex-

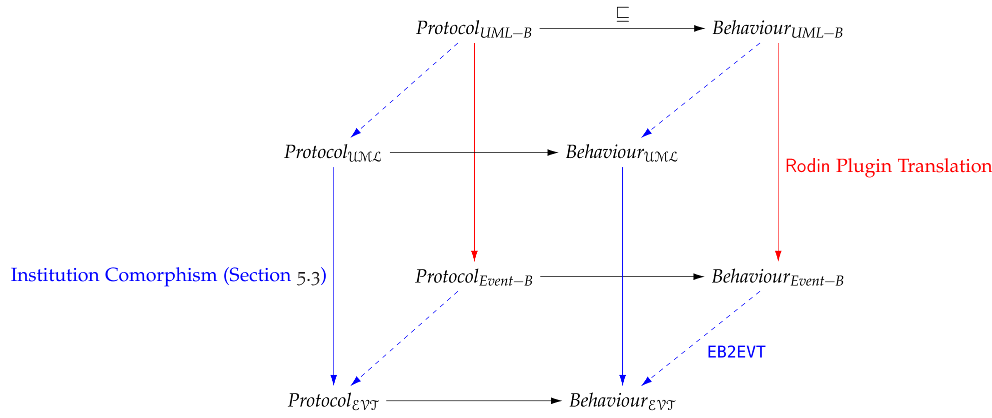


Figure 5.10: A refinement cube showing the various levels at which refinement takes place throughout the ATM example using the UML and Event-B formalisms and their respective institutions.

tending is not supported so we had to do this manually. If we use event extending and then add an auxiliary variable to match the state variable in the abstract machine, then refinement holds in the Event-B version with most of the proof obligations discharging automatically as can be seen in Figure 5.12.

A feature of our approach is the foundation it provides for refinement at different levels of specification. We illustrate the refinement relation in the various formalisms discussed in this paper using a refinement cube shown in Figure 5.10. The vertices in this cube represent specifications, where each specification is subscripted by the formalism that it is represented in.

The (lower) front face of the cube represents the institution- theoretic world of $\mathcal{UML}$ and $\mathcal{EVT}$, where the downward vertical arrows represent the translation via comorphism that we described throughout this paper. The (upper) back face represents elements in the Rodin environment,

```
1  CONTEXT mac0_implicitContext
2    SETS
3      protocol_STATES
4    CONSTANTS
5      Idle
6      Verifying
7    AXIOMS
8       typeof_Idle: Idle ∈ protocol_STATES
9       typeof_Verifying: Verifying ∈ protocol_STATES
10      distinct_states_in_protocol_STATES:
            partition(protocol_STATES, {Idle}, {Verifying})
11 END

12 MACHINE mac0
13   SEES mac0_implicitContext
14   VARIABLES
15     protocol
16   INVARIANTS
17      typeof_protocol: protocol ∈ protocol_STATES
18   EVENTS
19   Initialisation
20     then
21        init_protocol: protocol := Idle
22   Event verify ≙ordinary
23     when
24        isin_Idle: protocol = Idle
25     then
26        enter_Verifying: protocol := Verifying
27   Event verified ≙ordinary
28     when
29        isin_Verifying: protocol = Verifying
30     then
31        enter_Idle: protocol := Idle
32   Event reenterPIN ≙ordinary
33     when
34        isin_Verifying: protocol = Verifying
35     then
36        enter_Idle: protocol := Idle
37   Event markInvalid ≙ordinary
38     when
39        isin_Idle: protocol = Idle
40     then
41       Skip
42 END
```

Figure 5.11: The Event-B specification that was generated for the abstract protocol machine that was shown in Figure 5.9.

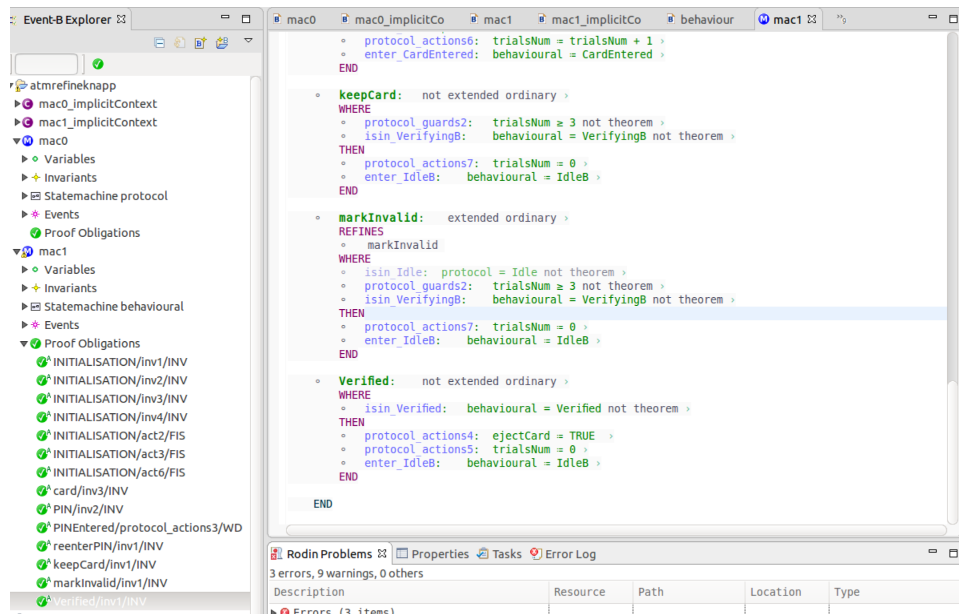


Figure 5.12: This figure illustrates that the proof obligations associated with the Event-B models produced are discharged automatically.

and in this case the downward vertical arrows represent the automatic translation provided by the UML-B plugin.

The horizontal arrows of Figure 5.10 indicate a refinement relation from an abstract to a concrete model or specification. That is, the leftmost face of the cube represents the four versions of the abstract protocol machine, and the rightmost face of the cube represents the four versions of the more concrete behavioural machine. For example, the topmost horizontal arrow represents the refinement in UML-B from Figure 5.9 to Figure 5.3.

5.6 SUMMARY

Our institution comorphism from the $\mathcal{UML}$ institution to $\mathcal{EVT}$ defines interoperability in a way that can also account for the functionality of the UML-B plugin. Thus, it provides a mathematically correct basis

for the translation between UML state machines and Event-B models. We discussed a number of approaches to defining this comorphism in Section 5.3.3 and analysed a selection of potential edge cases to our proof in Section 5.4.2.

Not only has this work exhibited the flexibility of our approach but it has enabled us to provide a mathematical underpinning for the UML-B plugin. We used the example of a simple ATM state machine to illustrate our approach throughout this chapter. Although the institution for $\mathcal{UML}$ has not yet been implemented in HETS (as outlined in Section 1.2), we anticipate that once it has been included it will be possible to implement this comorphism in HETS.

Moreover, by exploring the institution for simple UML state machines, $\mathcal{UML}$, and its interactions with $\mathcal{EVT}$, this work provides a new perspective to $\mathcal{UML}$. Since our comorphism allows the user to translate $\mathcal{UML}$ state machines to $\mathcal{EVT}$, it thus enhances these state machines with variant expressions which can be used to prove termination properties. Furthermore, the distinction between generated and consumed events in $\mathcal{UML}$ disappears in the comorphism, demonstrating that is not actually part of the essential structure of UML as captured by the $\mathcal{UML}$ institution.

The combination of specifications in Figure 5.10 captures the essence of the institution-theoretic approach to interoperability. As future work, we intend to further explore the possibilities provided by our framework. This would include investigating possible benefits of additional features deriving from the institutional approach, such as more flexible notions of refinement and modularisation constructs cross-cutting both formalisms. Similarly, the possibility of integration elements of Event- B with other UML diagrams, such as Event-B context with UML class diagrams, would greatly enhance the benefits of interoperability between these formalisms.

Part III

MODULARISATION

"Inside every large program, there is a small program trying to get out."

– C.A.R. Hoare

6

SPECIFICATION CLONES: AN EMPIRICAL STUDY OF THE STRUCTURE OF EVENT-B SPECIFICATIONS

The preceding chapters have examined the syntactic and semantic components of the Event-B language in great detail but we have not yet examined the size or complexity of Event-B specifications in practice. Therefore, we present an empirical study of Event-B specifications. Our study is exploratory, since it is the first study of its kind, and we formulate metrics for Event-B specifications which quantify the diversity of such specifications in practice [42]. In the setting of this thesis, this chapter motivates the need for standardised modularisation constructs in Event-B. We propose that the specification-building operators that are available in the theory of institutions can provide a solution.

6.1 BACKGROUND AND INTRODUCTION

We analyse Event-B specifications essentially as *software artefacts* and extend software engineering techniques to the Event-B language. We have approached this empirically, by assembling a large corpus of Event-B specifications and developing basic metrics to quantify their size and complexity. Since *refinement* is a key feature of the Event-B approach, we seek to quantify this aspect of Event-B specifications in particular, so we can understand how such refinement is carried out in practice [86].

As discussed in Section 2.2.1, apart from refinement, the modularisation constructs in Event-B are not well-developed, and a number of alternatives have been proposed to address this. As a contribution to the development of modularisation constructs for Event-B, we conduct

a study of *clones* in our corpus of Event-B specifications [98]. Studies of this kind already exist for software written in a variety of programming languages, but we believe this is the first time this topic has been addressed at the specification level.

There has been some previous work done on identifying suitable metrics for Event-B developments by Olszewska and Sere using the Halstead model [92]. The objectives of their work were to determine the size of an Event-B specification, the difficulty in constructing it and the effort required in designing and proving. The case study was limited to just one project with 7 machines, and it is not clear whether the Halstead metrics, dependent on defining and counting operations and operands, are the most appropriate way of characterising Event-B specifications in general.

6.1.1 CLONES IN CODE AND SPECIFICATIONS

The detection, analysis, management and tool evaluation corresponding to *code clones* represents a growing research area in the field of software engineering [98]. The reuse strategy indicated by code cloning can be beneficial in that it promotes the reuse of reliable code and can save time and effort in development. It is often the case, however, that duplicated code is caused by limitations in the programming paradigm's modularisation mechanisms and thus such code signals that improvements are required.

Roy et al. identify four different types of code clones [98], based on categorising the nature of the match between different pieces of code:

TYPE-1: identical code fragments that differ only in variations of white space and comments.

TYPE-2: structurally/syntactically identical code fragments that differ only in the names of identifiers, literals, types, layout and comments.

TYPE-3: a more liberal version of Type-2 clones which allow differences such as additions, deletions or modifications of statements.

TYPE-4: code fragments that exhibit the same functional behaviour but are implemented through very different syntactic structures.

In this chapter, we extend these definitions to detect clones between Event-B machines, contexts and events. Some work on identifying clones at the specification level has been done as part of the Arís project which retrieves reusable software artefacts using a graph matching approach [94]. However, this approach was based on finding matches in Spec#/C# code, and does not provide any data on the kind of clones found or their distribution.

6.1.2 MODULARISATION OF EVENT-B SPECIFICATIONS

In Section 2.2.1, we examined the current approaches to modularising Event-B specifications [60]. Since such modifications are made via Rodin plugin development, we enumerated the relevant modularisation plugins in Table 2.1. The use of these plugins can potentially reduce the number of clones in Event-B specifications, so we discuss them, where relevant, in our clone analysis results in Section 6.4. Furthermore, the specification-building operators can be used to modularise specifications and we illustrate how they can be used to declone Event-B specifications in Section 6.5. First, we analyse a corpus of Event-B specifications and support the claim that such modularisation constructs are required in practice.

Smaller Projects									
Project	Macs	Cons	Evs	Refs	Sens	Auto	Inter	RP	Vars
Bepiv6.4*	2	10	45	1	948	560	370	0	45
SSF_pilot	4	4	35	3	842	170	2	19	53
DynStabLSR	7	1	69	6	788	247	140	37	48
ch8circarbiter	6	2	46	5	764	153	0	31	57
TreeFilePerm	4	4	33	3	655	107	52	18	125
RCPert	4	3	53	3	583	199	28	29	18
RCNorm	4	3	49	3	565	146	32	27	18
ch912_mobile	6	1	43	5	539	134	19	19	34
ch917_train	5	3	38	4	539	128	5	23	35
SignalControl	10	4	106	9	497	135	0	26	100
ch7_conc	5	1	45	4	484	239	9	22	46
routing_new	8	4	51	7	479	226	60	47	26
FloodSet	6	5	27	5	445	209	87	46	16
ch2_car	4	3	34	3	438	249	4	17	29
ssf	7	4	51	6	430	48	11	8	57
seqpattern	5	2	30	3	425	37	1	4	33
SharedBuffs	4	1	22	3	423	98	5	19	24
SimpleLyra	4	4	28	3	418	55	0	3	24
gcd	7	3	32	6	407	91	84	21	82
ch8circpulser	9	0	52	7	397	93	1	20	36
ch916_doors	5	3	31	4	380	101	2	14	23
ch8circroad	5	0	27	4	379	37	0	9	47
ch8circight	3	0	19	2	370	89	0	25	30
ch6_brp	6	3	47	5	360	149	0	16	29
Modes_v2	3	3	30	2	333	108	13	3	30
aocs_t2	3	2	29	2	297	105	13	18	26
CtsCtrl	4	3	18	2	274	150	19	21	18
Rabin	7	7	62	6	262	138	71	2	41
pomc	5	3	27	4	257	81	27	10	27
pomcwoterm	5	3	27	4	250	83	13	10	27
ch913_ieee	6	3	21	4	243	71	21	16	18
DSAOCSSv3	1	1	9	0	233	82	8	0	8
AStyleQR	5	1	19	4	226	70	5	14	26
DSAOCSSv2	1	1	9	0	219	81	8	0	8
ch4_file_1	5	2	17	4	192	47	5	9	28
FindP_P1	4	1	21	3	191	27	15	13	18
aocs_t2_um	2	2	16	1	178	95	8	13	19
pat9QR	5	0	22	4	161	44	6	8	24
BinarySearch	3	1	14	2	154	102	6	13	8
SSF1	1	3	6	0	148	25	0	0	3
ch911_tree	5	3	15	4	140	81	0	9	8
SSF_minipilot	1	1	8	0	127	20	4	0	10
Club-120130	3	4	11	2	105	50	7	5	4
BoschSwitch	2	1	10	0	102	15	4	0	7
ch915_sort	3	1	12	2	101	56	11	11	8
ex-bubblesort	2	1	7	1	98	46	6	7	4
FindP_D	2	2	8	1	98	47	2	5	6
FindP_G	1	1	6	0	94	0	0	0	5
program2	2	2	9	1	88	192	5	3	5
Zer_ess	0	5	0	0	88	40	15	0	0
ex-bubbles	2	1	8	1	83	41	1	13	4
HermanRing	2	3	8	1	82	35	22	2	7
cae-square	3	3	10	2	78	53	1	2	8
primrec	2	2	7	1	74	36	0	2	4
FindP_P2	1	1	4	0	73	0	0	0	3
ch915_bin	3	1	11	2	68	32	5	7	7
AStyleQR_2	1	1	5	0	65	10	0	0	7
TrafficLights	2	1	11	1	58	20	0	0	7
ch915_inv	2	2	7	1	55	32	0	5	3
Cowboy	2	1	7	1	53	14	1	1	4
ch910_ring	2	2	6	1	52	24	4	1	3
ch915_sqrt	3	1	9	2	44	17	0	5	5
ch915_rev	2	1	6	1	43	28	3	4	4
ch915_mini	2	1	6	1	42	24	1	1	4
DiningCrypt	3	1	6	1	42	21	3	0	14
AStyleQR_3	1	1	3	0	40	6	0	0	4
AStyleQR_1	1	1	3	0	37	5	0	0	4
pat8SynMC	2	0	5	1	34	15	0	0	4
ch915_search	2	1	6	1	29	17	0	3	2

Table 6.1: Metrics for the Event-B projects that fall into the "smaller" category.

Legend for column headings for Tables 6.1, 6.2 and 6.3
Macs: # of machines
Cons: # of contexts
Evs: # of events
Refs: # of refinement steps
Sens: # of sentences
Auto: # of automatic proofs
Inter: # of interactive proofs
RP: # of designated refinement proofs
Vars: # of variables

Larger Projects									
Project	Macs	Cons	Evs	Refs	Sens	Auto	Inter	RP	Vars
Midas*	43	61	2500	40	26395	2034	3163	2183	207
FlashFileFS	18	6	320	13	5442	974	531	88	343
DepSatSpec	14	2	2094	13	4771	1309	549	0	1811
ATM	7	12	129	6	3447	925	37	46	150
B2Bminip	12	0	228	11	2900	425	73	124	45
Bepiv3.3	6	6	137	1	2665	153	113	12	311
TSHHDMac	35	50	1487	18	2661	602	84	15	782
Bepiv5.0	9	10	329	8	2007	683	317	0	141
CDIS	7	6	103	6	1894	101	0	3	301
HDMac	19	25	718	16	1605	448	23	2	396
Pilot_v3	4	4	98	3	1586	134	9	0	190
MLLanding	11	2	313	10	1432	286	210	0	277
FlashFileFL	6	12	109	5	1243	379	13	11	98
HLanding	11	7	321	9	1213	173	68	17	173
ch3_press	8	3	144	7	1200	0	0	0	70
OnbCont	9	3	224	8	1108	438	1	14	158

Table 6.2: Metrics for the Event-B projects that fall into the "larger" category. Outliers are indicated by an asterisk*.

All Projects ($n = 85$)									
Project	Macs	Cons	Evs	Refs	Sens	Auto	Inter	RP	Vars
Minimum	0	0	0	0	29	0	0	0	0
Median	4	2	27	3	274	83	5	8	24
Maximum	43	61	2500	40	26395	2034	3163	2183	1811
MADN	3.0	1.5	28.2	3.0	298.0	86.0	7.4	11.9	28.2

Smaller Projects ($n = 69$)									
Project	Macs	Cons	Evs	Refs	Sens	Auto	Inter	RP	Vars
Minimum	0	0	0	0	29	0	0	0	0
Median	3	2	17	2	192	56	5	8	18
Maximum	10	10	106	9	948	560	70	47	125
MADN	1.5	1.5	16.3	1.5	206.1	54.9	7.4	11.9	17.8

Larger Projects ($n = 16$)									
Project	Macs	Cons	Evs	Refs	Sens	Auto	Inter	RP	Vars
Minimum	4	0	98	1	1108	0	0	0	70
Median	10	6	270	8	1950	431	70	11	242
Maximum	43	61	2500	40	26395	2034	3163	21.83	1811
MADN	5.2	5.9	203.9	4.4	1076.4	398.1	97.1	17.0	130.5

Table 6.3: Summary statistics for the whole data set, and for the two "smaller" and "larger" subdivisions.

6.2 ANALYSING A CORPUS OF EVENT-B SPECIFICATIONS: METRICS AND REFINEMENT

Considering that there has been no previous large scale study in this area, we focus on conducting an exploratory data analysis to identify and quantify the main characteristics of Event-B specifications.

In order to carry out this analysis we assembled a corpus of Event-B specifications. We obtained the Event-B projects in this corpus from a number of publicly-available Event-B resources, including the Event-B Wiki Page, the DEPLOY website and the case study tracks at the ABZ conference (2014 and 2016). Some additional projects were obtained directly from the developers who constructed them. In total we obtained 85 Event-B projects, ranging from smaller textbook-style examples through to large-scale developments.

All of the specifications in these 85 projects could be processed using the Rodin platform, and were thus available as a set of XML files in a standardised format. To analyse these projects we developed a suite of Python programs that read in the files in Rodin format, calculated and reported metrics, and searched for occurrences of clones at various levels[1].

6.2.1 QUANTIFYING SPECIFICATION SIZE

The most obvious measurable entities in an Event-B specification correspond to the major syntactic categories. Just as the size of a software project might be measured using code metrics such as number of classes, methods or lines-of-code, we can get similar information from an Event-B specification in terms of the number of contexts/machines, events and

1 More details on the implementation of our metric and clone detector can be found in Appendix B.

sentences. Specific to a formal approach, we can also measure the number of proof obligations (automatically and interactively proved). The metric values for the 85 projects in the corpus are given in Tables 6.1 and 6.2.

In total, for all 85 projects in the corpus there are 359 contexts and 468 machines, which in turn contain 10828 events. One immediate difficulty in analysing the corpus is the overall range of the specifications, from small, textbook-style examples, through to major systems. We chose to divide the corpus based on the *number of sentences* (these include axioms, invariants, variants, guards, witnesses and actions) per project, since this was the metric closest to lines-of-code, which might best reflect a simple measure of size for a project. Thus the rows of Tables 6.1 and 6.2 are ordered based on the total number of sentences in a project. We note that this is a coarse-grained measure as sentences may vary in complexity. This level of complexity could be evaluated by converting predicate sentences into conjunctive normal form (CNF) and counting the number of clauses.

In order to be able to represent this information meaningfully and extract useful information from it, we have split the corpus into two different data sets. We investigated a variety of ways by which to carry out this split, including:

- using the examples from the *Modeling in Event-B* textbook [3] as models of "smaller" projects, and regarding projects with more sentences than these as "larger" projects.
- extracting the outliers using Tukey's test (the median plus 1.5 times the inter-quartile range); all such outliers were larger projects.
- using trimming [71], to identify a fixed proportion at the extreme ends of the data set.

In practice, these three strategies resulted in almost the same set being identified, and we have used Tukey's test to categorise the 16 projects

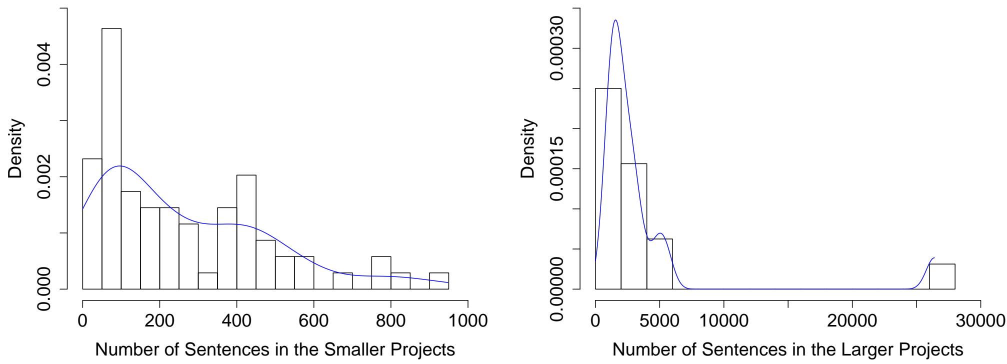


Figure 6.1: Histograms showing the distribution of the numbers of sentences per project for the smaller and larger data sets. Note that the vertical axes here are on different scales.

in Table 6.2 as "larger". This also corresponds to the top 19% of the projects, and excludes all but one of the textbook examples (the exception is the mechanical press controller from Chapter 3 of Abrial's book [3]). We refer to the 69 remaining projects listed in Table 6.1 as "smaller". These projects all have 10 contexts or under and 10 machines or under. Some of these are non-trivial projects however, and the number of sentences ranges from just 29 up to 948. Thus we have further divided Table 6.1 into quartiles based on the number of sentences.

Tables 6.1 and 6.2 demonstrate the diversity of Event-B developments and we provide them so that future studies have a measure with which they can gauge the comparative size of Event-B developments.

6.2.2 METRICS FOR EVENT-B SPECIFICATIONS

Figure 6.1 further illustrates the diversity in size between the projects, showing the distributions of the sentences in the smaller and larger project sets. These measurements signal that one should be cautious when choosing a representative Event-B specification as the structures

vary so much. In particular, the Midas project, which specifies an Instruction Set Architecture (ISA) that gets refined to a usable Virtual Machine (VM), is a dramatic outlier of this data set on almost all metrics, as is shown by the rightmost bar in Figure 6.1, and thus should be considered quite distinctive as an Event-B specification.

Table 6.3 summarises the ranges for each of the metrics, giving the minimum, maximum, median and MADN values for the whole data set and its two subdivisions. Due to the uneven distribution we use the median and MADN as robust measures in place of the mean and standard deviation. MADN is the median of the absolute deviations from the median, divided by $z_{0.75}$ [71]. It is notable that in most cases the MADN is close to or exceeds the median, indicating a large spread of values for each of the metrics.

We analysed all of the metrics in Tables 6.1 and 6.2 to check for inter-relationships, using Spearman's rank correlation coefficient. The most notable very strong correlations (with $p < 0.001$ in all cases) were between the following variables:

- **the number of events and the number of sentences** in the small data set ($\rho = 0.905$), where the median number of sentences per event is 11 (MADN $= 4.4$). However, in the larger project set, this correlation is weak ($\rho = 0.391$). The larger projects contain a greater number of contexts, thus adding sentences to the projects that are not sentences within events.
- **the number of machines and the number of events** in both the smaller ($\rho = 0.849$) and larger ($\rho = 0.904$) project sets. The median number of events per machine is 25 (MADN $= 9.8$) in the larger set and 5 (MADN $= 2.7$) in the smaller.

There was also a (lower) strong correlation in the smaller projects between the numbers of events/sentences and the number of automatic proofs. Throughout the project set as a whole, there was a particu-

larly strong correlation between the numbers of events and variables ($\rho = 0.935, p < 0.001$), and also between the numbers of sentences and variables ($\rho = 0.919, p < 0.001$). This indicates that as the size of the state (represented by variables) increases so do the numbers of events and sentences. New variables are often added during refinement steps, as such, we found a strong correlation between the numbers of machines and variables ($\rho = 0.821, p < 0.001$). Thus we expected the strong correlation between the numbers of machines and variables that was discovered during our analysis ($\rho = 0.857, p < 0.001$).

The data in Tables 6.1 and 6.2 shows that the number of automatic proofs required dramatically exceeds the number of interactive proofs in general. On average, in the larger projects, 78.6% of the proofs were done automatically with 91.1% of the proofs automatic in the smaller projects. This is important for automated verification, since it is a measure of the relative amount of theorem-proving work imposed on the user, as compared to that done by the underlying prover. It is notable that this percentage is much higher for the smaller examples than for the larger ones. This is most likely due to the increased complexity in modelling large-scale systems. As Event-B continues to be used industrially, this metric can be useful in measuring the degree to which automated theorem-proving has increased in effectiveness.

6.2.3 QUANTIFYING REFINEMENTS

Figure 6.2 contains a histogram with kernel distribution, showing the number of refinement steps for each of the project sets. As can be seen, in the larger project set the Midas project is again a dramatic outlier with 40 refinement steps. The smaller project set does not contain any dramatic outliers, with approximately 50% of these projects containing only one refinement step. We postulate that this is due to the "smaller"

project set containing specifications that have been designed for teaching.

In both the smaller and the larger project sets there is a very strong correlation between the number of machines and the number of refinement steps in a project ($\rho = 0.989$ and $\rho = 0.993$ respectively, $p < 0.001$). In most cases the relationship is almost 1:1, showing that *linear* refinement chains are the most common refinement strategy used. By default, a machine can refine at most one other, so typically a machine will have one *parent*. These refinement chains bear a striking similarity to the notion of refinement presented in the theory of institutions which is typically a single, linear chain [100]. While the *Feature Composition* plugin for Rodin allows the merging of machines in a refinement step [54], this is clearly not the usual approach taken in these examples.

In Event-B, proof obligations are one indicator of the complexity of the system being modelled. There is a specific set of proof obligations that are generated through the refinement of events (guard strengthening and merging, action simulation, equality of a preserved variable, witness well-definedness and witness feasibility). We list the number of these designated refinement proofs in the rightmost column of Tables 6.1 and 6.2. These proofs are only generated for refined events that are labelled as `not extended`. Events that are labelled as `extended` generate no proof obligations that are designated for refinement as they are specific to superposition refinement. This is quite an efficient approach to refinement as Rodin avoids the regeneration of these proofs [6], but is only applicable where no data refinement has taken place.

There is a strong correlation between the number of refinement proofs and the number of refinement steps in a project in the smaller project set ($\rho = 0.786$, $p < 0.001$) resulting in the median ratio of 3 refinement proofs to 1 refinement step. However, the correlation is not significant for the larger project set. We found that developers of the larger projects

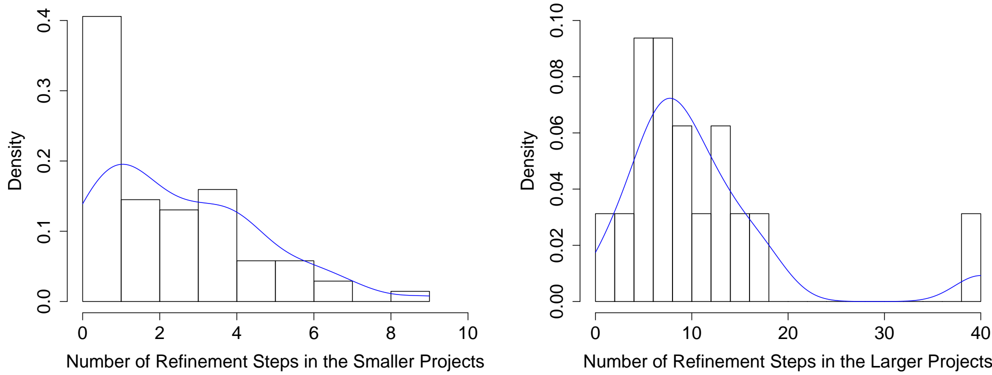


Figure 6.2: Histograms with kernel distribution describing the number of refinement steps taken in both the smaller and larger project sets. Note that the vertical axes here are on different scales.

often opted to avoid data refinement and use event extending to streamline their developments. Based on the data in Table 6.2 we can identify 5 out of 16 projects (31.25% of the "larger" project set) that used this approach.

We had expected a correlation between the number of refinements and the number of sentences, with machines increasing in size as they became more concrete. However, this correlation is not strong even in the smaller data set ($\rho = 0.695$, $p < 0.001$) and neither strong nor significant in the larger, which, as mentioned earlier, are also influenced by the large number of contexts.

6.3 DETECTING SPECIFICATION CLONES

In this section we describe our strategy for applying the clone types discussed in Section 6.1.1 to Event-B.

In all cases we will be comparing *sentences* from one specification with those in another: this includes axioms in contexts, invariants and variants in machines, and guards, witnesses and actions in events. There

are a number of approaches to matching in the literature, including metric, token, text and abstract syntax comparison between statements [14]. In keeping in line with the literature which uses statements, we have used sentences, which are the nearest comparable entity to statements, as the smallest unit of matching. All sentences are tokenised to eliminate formatting and white-space, and we compare only sentences of the same kind (thus axioms with axioms, etc.). We have discounted any machines/contexts/events with 2 or less sentences in order to ensure that we are only collecting meaningful clones.

We carry out this matching at three levels: contexts, machines and events. The order of statements in an Event-B specification does not matter [3]. However, the order of statements is important in the code cloning literature and so we use sequences of sentences in our approach. We base our search for clones on the clone types discussed in Section 6.1.1. In all cases, (context, machine and event):

- Type-1 clones correspond to exact matches between the full sentence sequences in each case: that is all sentences in one component must match all those in the other.
- Type-2 clones are matches between the full sentence sequences, but where variable names are anonymised, each variable name being replaced by a positional indicator.
- Type-3 clones are also matches between two sentence sequences (with variable names anonymised or unanonymised), except that matches between *sub-sequences* of the sentences are now allowed. We calculate the percentage of type-3 clone similarity using the maximum of the similarity calculated for both the anonymised and unanonymised versions.

We do not explicitly search for type-4 clones (functional equivalence) in what follows. From one perspective, all of our clones could be viewed as type-4, since we are not really comparing code but specifications, and

thus identifying a degree of functional equivalence. However, a more robust search for type-4 clones would require proof of the equivalence of the corresponding generated proof obligations for machines, contexts and events, which we have not attempted. As such, we omit type-4 clones from further discussion here as future work.

We have conducted an automated analysis of our corpus of projects by writing a series of Python scripts that read in the Rodin files, represent the components as an abstract syntax tree, and then perform comparisons at the context, machine and event level. We analyse machines and events both with and without any corresponding variants and invariants included, to distinguish between sentences that are global and local (to events) in the machine. Variants are only included with events that have a status of `anticipated` or `convergent`, since, unlike `ordinary` events, these are required to not increase and respectively decrease the variant expression [3].

We have also identified the clones that occur most frequently throughout our corpus, at the level of machines, contexts and events. As there are no libraries for Event-B specifications and since contexts typically supply custom data types, we were interested in examining whether or not similar contexts have been used in the Event-B projects across our corpus. Thus we also determine whether the clones that we have discovered are *inter*-project (across different projects) or *intra*-project (within the same project) clones.

6.4 results of the clone analysis

In this section we summarise the results of our clone analysis through the entire corpus. In what follows we regard the three clone types as mutually exclusive: by type-2 we mean all those that are type-2 but not type-1, and by type-3 we mean those that are type-3 but not type-1

or type-2. Table 6.4 summarises the results of this analysis, providing counts for the number of clonings identified (type-1, type-2 and type-3) and also the number of clones. Our analysis returns pairs corresponding to instances of cloning that have occurred. We refer to these as *clone pairs* or *clonings* in what follows.

6.4.1 CONTEXT CLONES

As can be seen in the first row of Table 6.4, our analysis found 40 clone pairs at the context level in the corpus, consisting of 18 type-1 and 22 type-3 clone pairs. We had expected this, since contexts resemble data types in a programming language. The *theory plugin* offers a potential solution to this problem as it provides a way of adding new data types to Rodin [18].

When we investigated the actual clones that were returned we found 22 context clones, of which 18 occurred on an inter project basis and 6 on an intra project basis. There were 2 which occurred both as inter and intra project clones. The fact that so many of them occurred between different projects supports our claim that they are being re-used in a manner similar to libraries. The inter project clonings occurred mostly between projects that shared a common approach or between projects that were modelling the same kind of system. For example, there were quite a few inter project clonings in the separate developments of a Hemodialysis Machine [61, 80], the different versions of BepiColombo [75], and the assortment of file systems being modelled (Flash FS, Flash FL and Tree FS) [3, 24, 25].

Event-B	Clone Pairs				Actual Clones	
Component	Type-1	Type-2	Type-3	Total	Total	Occur.
Contexts	18	0	22	40	22	51
Machines	13	7	937	957	19	40
Machines (+VI)	9	7	943	959	13	28
Events	276	942	4781	5999	131	417
Events (+VI)	35	158	7229	7422	65	175

Table 6.4: The occurrence of clone pairs and clones per type throughout the entire corpus. Note that '(+VI)' indicates that the variants (where appropriate) and invariants have been included in the analysis.

6.4.2 MACHINE CLONES

In Event-B, a machine is generally reused by means of refinement and thus we did not expect to find many type-1 clonings or inter project clones. As can be seen in the second and third data rows of Table 6.4, we discovered a very small number of type-1 and type-2 machine clonings. We did, however, manage to identify 937 type-3 clone pairs in the analysis without the variants and invariants included.

Since the type-3 clone pairs are identified in terms of their similarity, expressed as percentages, we provide an illustration of the distribution of type-3 clones in Figure 6.3. The top two histograms in Figure 6.3 show the data for machine-level clone pairs, and the bottom two for event-level clone pairs. As expected, the distributions for machine-level clones skew to the left, as most clones had a low similarity percentage, indicating that there is some basic machine structure being reused over and over again but the part that is being cloned does not contain a

large proportion of the sentences. Nonetheless, there is still a significant number of clone pairs that have at least 50% of their sentences matching.

In total we found 5 inter and 14 intra project full machine clones. This reduced to 3 inter and 10 intra project clones when the variants and invariants were included. Most of these were within the same project and therefore were most likely caused by refinement chains. These numbers are quite small with regards to the size of our corpus, thus we conclude that full machines typically do not incur a huge amount of cloning. In cases where this form of cloning occurs, the pattern plugin can be used as a way of modularising the specification [47].

6.4.3 EVENT CLONES

Since events are the smallest unit of modularisation, we expected a higher level of cloning to be found between pairs at this level. The fourth data row of Table 6.4 shows that we identified 276 type-1, 942 type-2 and 4781 type-3 clone pairs or instances of event clonings in our corpus. As can be seen from the fifth data row in Table 6.4, this number decreased for type-1 and type-2 when we included the appropriate variants and invariants (35 and 158) respectively. The number of type-3 clone pairs, however, increased quite dramatically to 7229. This is because the inclusion of variants and invariants increased the size of many small events beyond our threshold of 2 sentences, thus including events in the analysis that were absent when these variants and invariants were not included. For events that contained a small number of sentences before the inclusion of variants and invariants, this corresponded to a crude search for variant and invariant clones.

There were 131 different event clones, of which 30 were inter and 126 were intra project clones. Intra project clonings occurred 382 times and they occur in the scenarios where one event is refined throughout

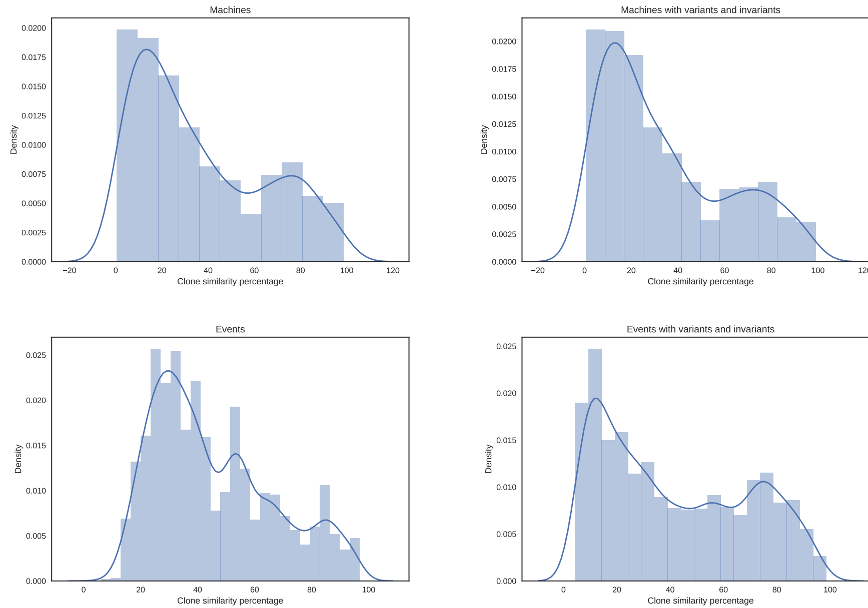


Figure 6.3: Histograms describing the distribution of Type-3 clones across the entire corpus of Event-B specifications. Note that we have omitted type-3 *context* clones as there were relatively few of these.

a project and also where there are event clonings within the same machine. We found 210 situations where one event in a machine was a clone of another event in the *same* machine. This accounts for approximately 1.9% of the total events in our corpus and 17.2% of the total type-1 and type-2 event clone pairs. Inter project clonings occurred a total of 37 times.

Based on this analysis, we conclude that there may be a relationship between the number of intra event clones between different machines in the same project and the level of refinement of that project. However, this needs to be examined in more detail.

6.5 DECLONING EVENT-B SPECIFICATIONS

One way of addressing the large number of type-2 clones at the event level is through the provision of facilities for event re-use. This could

be done either through a renaming feature as a Rodin plugin, or by introducing parmetrisation constructs at the Event-B language level.

The *renaming refactory* plugin could offer some assistance here as it renames components of an Event-B model with the renamings propagating through to the proof obligation level. However, it does not offer any way of instantiating copies of events. The *Pattern* and the *Generic Instantiation* plugins are also relevant, but these currently work only at the machine level, rather than the event level [47, 108].

If more sophisticated modularisation constructs were made available for Event-B, they could potentially alter the development strategy taken by developers and turn what would have been type-3 clones into type-2 clones which could be parametrised and then added to in future refinements.

Decloning results in more modular specifications and thus allows us to avoid replicating specifications. In situations where bugs are present in the original artefact then these bugs will be present at all instances where it is reused whether it is by copy-and-paste or by importing modules. A modular system that facilitates importing modules enables the developer to only repair the original module whereas the copy-and-paste approach means that all copied code must be mended. Clearly, modular development is beneficial, however, there are some dangers lurking when modules are imported/reused. For example, the developer must ensure that any software artefacts that are to be reused behave as expected and are free from bugs. If the developer does not abide by this then the resultant system may not behave as expected.

Throughout this thesis, we proposed the theory of institutions as a mathematically sound framework to incorporate Event-B into and thus provide users of Event-B with access to an array of generic and formalism-independent modularisation constructs through the use of specification-building operators [40]. These specification-building oper-

```
1 spec  State =
2   sort States
3   ops s1, s2, s3, s4 : States
4 end
```

```
5 spec  State1 =
6   State with States ↦ TopLevel_States,
7     s1 ↦ TL_Standby, s2 ↦ TL_Preparation,
8     s3 ↦ TL_Initiation, s4 ↦ TL_Ending
9 end

10 spec  State2 =
11   State with States ↦ LowLevel_TestingCF_States,
12     s1 ↦ LL_Testing_CF_Untested,
13     s2 ↦ LL_Testing_CF,
14     s3 ↦ LL_Testing_CF_KO,
15     s4 ↦ LL_Testing_CF_OK
16 end
```

Figure 6.4: Decloning the context clone that appeared on an intra and inter project basis throughout the Hemodialysis Machine developments.

ators can be used to declone Event-B specifications. We illustrate this with respect to context, machines and events in what follows.

6.5.1 DECLONING CONTEXTS

In Section 6.4.1, we established that context clones occurred frequently throughout our corpus. In particular, there was one context that was cloned four times throughout the developments of the Hemodialysis machines. It describes a set of states and can be represented as an $\mathcal{EVT}$-specification as shown in Figure 6.4 (lines 1–4). We have illustrated how two of the clones in the Hemodialysis development can be modularised by instantiating two copies of this clone using the `with` specification-building operator to rename their components.

After further investigation, we observed that this context was also a clone of 11 others throughout our corpus.

```
1 spec AbstractMac =
2   sorts s1
3   ops o1
4   . p1
5   Events
6     Initialisation
        ⋮
7
8 end
```

```
9 spec ConcreteMac =
10   AbstractMac
11     then
12       sorts s2
13       ops o2
14       . p2
15       Events
16         Initialisation
             ⋮
17
18 end
```

Figure 6.5: This figure illustrates how superposition refinement between an abstract and a concrete specification can be represented using the then specification-building operator. Here the two Initialisation events are merged as they have the same name.

6.5.2 DECLONING MACHINES

In Section 6.4.2, we observed that machine clones tend to occur throughout refinement chains. This occurred quite frequently throughout the refinement steps of the ATM machine. The specification-building operators allow us to avoid respecification during refinement steps by using the then operator. Figure 6.5 shows how an abstract machine can be included in a more concrete machine that exhibits superposition refinement. In this scenario, any events with the same name are merged.

6.5.3 DECLONING EVENTS

Our results show that event clones occurred frequently throughout our corpus. For example in one of the machines of the DepSatSpec Event-B specification (Altitude and Orbit Control System), the same event that contained nine sentences and three parameters was cloned (type-1) three times. These three events refined the same abstract event and the superposition refinement that was added was exactly the same in all

```
1  spec  EventSpec =
2    DataSpec
3    then
4      Events
5        Event AocsMgrEvent ≙ordinary
6          any  MoveTo, branch, unit
7          when  ActiveMgr = AocsMgr
8                Cursor ≠  0
9                MoveTo ∈  ℕ
10               branch ∈ BRANCHES
11               unit ∈ UNITS
12         thenAct  Cursor := MoveTo
13                  HW_Simulation_Receive_Command_Assertion_Br_Status
14                          := bool(UnitMgr_Units_Br_Status(unit ↦branch) = BrStatus_Locked)
15                  HW_Simulation_Receive_Command_Assertion_BrId := branch
16                  HW_Simulation_Receive_Command_Assertion_UnId := unit
17 end

18 spec  CombineEventSpec =
19   EventSpec with AocsMgr Event ↦ Call_HW_Simulation_Get_Sensor_Data and
20   EventSpec with AocsMgr Event ↦ Call_HW_Simulation_Spurious_LOA_Error and
21   EventSpec with AocsMgr Event ↦ Call_HW_Simulation_Receive_Command
22 end
```

Figure 6.6: Decloning the events in the DepSatSpec Event-B specification using specification-building operators.

three cases. This Type-1 cloning could have been avoided by specifying the event once and using the specification-building operators (`with`, `and`) to combine three versions of the event specification. This is illustrated in Figure 6.6.

6.6 THREATS TO VALIDITY

One feature of our work is the creation of a corpus of Event-B projects, and our division of this set into smaller and larger projects. The selection poses a threat to *conclusion validity*, since we are dealing with a heterogeneous group of projects, and there is a risk that the differences in metrics are due to other factors not measured here, such as heterogeneity in terms of the domain of application, e.g. railway, health-care, control systems, algorithms, etc.

Our analysis of the projects is conducted based on the metrics that we have defined and measured. While these metrics corresponded to ma-

jor syntactic categories in Event-B and have clear analogies with similar constructs in programming languages, there is a threat to *construct validity* here. In particular, further studies would be required to establish the predictive value, if any, of these metrics.

Similarly, in adapting the definition of code clones to Event-B we made a number of decisions on what should be measured and the degree of matching involved; altering these could yield different results. Our measurement of type-3 clones was based on sentence sequences and the in-order anonymisation of variables: a more general technique could produce more clone-pairs, at the cost of a considerable increase in combinatorial matches.

Since our analysis was based on processing the XML files generated by Rodin, we have a high degree of confidence that the measurements are accurate, and do not pose a threat to the *internal validity* of our results. However, in three of the Event-B projects (`ch3_press`, `FindP_G` and `FindP_P2`) the corresponding .bps files, which hold information about the proofs, were empty. Thus these projects have no automatic or interactive proofs recorded even though proof obligations have been generated. We believe that these projects may have used an older version of Rodin or a plugin that we do not have access to. One approach to resolving this would be to remodel them using a current version of the software with no extra plugins installed. We chose not to do this as we wished to remain as impartial as possible with regard to the corpus that we collected.

In total, there are 85 Event-B projects in our corpus, but it is possible that this is not a large enough sample size to study. This causes a threat to *external validity* in terms of the generalisability of our results. We believe that assembling and maintaining a measured corpus of Event-B programs is a worthwhile task in this regard.

6.7 SUMMARY AND FUTURE WORK

This chapter expands on the clone study that was published in [42] by including the measurement of the size of the state throughout our corpus of Event-B specifications. We have also illustrated how Event-B specifications can be decloned using the specification-building operators of the theory of institutions.

Our work applies the existing software engineering approaches of calculating metrics and detecting code clones to specifications written using the Event-B formal specification language. This exploratory study is the first of its kind and has enabled us to provide and analyse the metrics of a corpus of Event-B specifications. In this way, we provide a benchmark against which other Event-B projects can gauge their comparative size and complexity level. Our empirical study supports the claim that there is an underlying requirement for modularisation constructs in Event-B by evaluating code clones at the specification level.

Moreover, we have shown how the institutional framework can accommodate the kinds of modularisation required to reduce the number of specification clones at context, machine and event level. This is something that has not been achieved by any single Rodin plugin that is listed in Table 2.1. Thus, the institutional framework that we have incorporated the Event-B language into, is in fact, more accommodating to modularisation than the current state-of-the-art for Rodin.

Part IV

CONCLUSIONS

"Simplicity is prerequisite for reliability."

– Edsger W. Dijkstra

7

CONCLUSIONS AND FUTURE WORK

In this chapter we discuss potential avenues for future research and summarise our contributions.

7.1 FUTURE WORK

The research presented in this thesis has inspired a number of directions for future research and these are discussed here.

With regard to metrics and specification clones, future work includes the assessment of *clone genealogies*, particularly in the context of refinement – i.e. how clones evolve throughout successive refinements. This study would show us whether or not clones persist in the specification after it has undergone a (series of) refinement step(s). We are also interested in detecting non-typing invariant clones, this would allow us to analyse data refinement clones using gluing invariants. The application of this study to other formalisms used in the software verification domain would allow us to compare them with the results obtained in Chapter 6 and thus analyse specification clones in other specification languages. This could potentially include incorporating this clone detector into a general framework for detecting clones, such as PMD[1].

To our knowledge, the Heterogeneous Toolset, HETS, is the only available tool support for integrating and providing modularisation support for formalisms/logics defined as institutions. Of course, HETS does not provide a way of "prototyping" an institution, as it has no explicit way

1 http://pmd.sourceforge.net/pmd-4.3.0/cpd.html

of encoding the category of models or the satisfaction relationship of a particular institution. In the future, developing a tool with this functionality would make the mathematical theory of institutions more accessible to researchers and software developers. One possible approach would be to use a theorem prover with theories for category theory that could facilitate proving the validity of institutionally defined formalisms/logics such as Coq [56].

Our research has shed light on the benefits of the institution-theoretic approach and future work consists of incorporating other such formalisms into this framework in order to investigate their semantics and formally define interoperability between them. Unified Theories of Programming (UTP) provides a unified framework where program semantics are represented as theories. The central concept of UTP is that of a "theory supermarket" where one can shop for theories with confidence that they will work together [46]. This framework is not as mature as the theory of institutions, and is, in fact based in lattice theory [62]. We anticipate that UTP can be encapsulated in terms of institutions in future work.

7.2 SUMMARY

In a document describing the "Justifications for the Event-B Modelling Notation", Hallerstede remarked that the

> *"...duplication of concepts should be avoided and each single concept should have a single and unambiguous interpretation" [58].*

when referencing any extensions to the Event-B language and tool support. Unfortunately, this guideline has not been adhered to. This is evidenced by the sheer volume of approaches to providing modularisation (Table 2.1) and interoperability (Table 2.2) features for the Event-B specification language.

We have proposed the theory of institutions as a framework that provides a unified set of modularisation constructs and gateways to interoperability for specification languages such as Event-B. In Chapter 3, we incorporated Event-B into this framework by defining its institution, $\mathcal{EVT}$, [40, 41]. Until now, Event-B was not equipped with a formal semantics and in Chapter 4, we formalised a translational semantics from Event-B to $\mathcal{EVT}$ and bridged the gap between the Hets and Rodin software ecosystems using the EB2EVT tool that we developed.

By representing Event-B specifications in the institutional setting of $\mathcal{EVT}$, we were able to define and analyse interoperability between Event-B and UML. This interoperability was illustrated in Figure 1.1 and described in Chapter 5. Not only has this work exhibited the flexibility of our approach but it has enabled us to provide a mathematical underpinning for the UML-B plugin (Chapter 5).

In Chapter 6 we demonstrated the need for a unified set of modularisation constructs in Event-B via our empirical study that identified metrics and specification clones among a corpus of Event-B specifications. This has contributed a means for quantifying the size of Event-B specifications and we anticipate that it can be used by others to gauge the comparative size of their Event-B specifications. We also describe how some of the current Event-B modularisation techniques can be subsumed by the institution-theoretic specification-building operators, thus administering a more unified approach to modularisation in Event-B.

Improvements are generally made to the Event-B language and its tool support by building plugins for Rodin. Our research provides a solid grounding for Event-B using the theory of institutions that allows the developer to avoid building the nth plugin for Rodin in order to facilitate modularisation and interoperability for Event-B. Thus adhering to Hallerstede's guideline as quoted above.

By defining $\mathcal{EVT}$, we have provided a new perspective to the normally logic-oriented nature of institutions in the guise of an industrial-strength specification language. Moreover, by incorporating Event-B into this framework we were able to achieve our objectives and thus provide a more generic and powerful way to write and reason about (modular) Event-B specifications. Refinement is central to the Event-B methodology and we examined it under specific headings at various intervals throughout this thesis.

As described in Chapter 1, the formal methods landscape is abundant with languages and tools that are very useful for proving the correctness of software systems, however, the sheer volume of them is cause for concern in terms of formalising correct interoperability between them. We have shown that the theory of institutions provides a generic framework within which all of these formalisms can be represented and their interoperability expressed in a provably correct way. Therefore, this thesis acts as a manual for those wishing to incorporate formalisms similar to Event-B, such as TLA+ for example, into the institutional framework.

A

DECLONING EVENT-B SPECIFICATIONS USING SPECIFICATION-BUILDING OPERATORS

In this appendix, we illustrate how the functionality of each of Rodin modularisation plugins that were discussed in Chapter 2 (Table 2.1) can be captured using the specification-building operators and parametrisation made available in the theory of institutions using $\mathcal{EVT}$. The clones study that we presented in Chapter 6 has motivated the need for modularisation constructs in Event-B. Here, we not only provide a theoretical foundation for current modularisation approaches in Event-B but we deliver a unified approach to modularisation in Event-B and discuss refinement in this setting.

A.1 RODIN PLUGINS

Chapter 2 introduced a series of plugins that have been developed for the Rodin Platform which offer modularisation features for Event-B specifications (Table 2.1). Here, we provide a brief overview of each of them and illustrate by example how their functionality can be captured using the institution-theoretic specification-building operators and parametrisation of specifications that were described in Section 2.4.1.

A.1.1 FEATURE COMPOSITION

The Feature Composition plugin was last modified in 2010 and works on *Rodin* version 2.0. It was inspired by work on the composition of Event-B models [95]. A *feature* is comprised of a machine and its (seen)

contexts. The objective is to combine these features by *fusing* variables and events. *Variable fusion* is achieved by concatenating the variable lists of specific features whereas *event fusion* is a mechanism of merging events with the same name. Event fusion corresponds to conjoining the guards of two events that share the same name and using parallel composition to combine their actions. The Feature Composition plugin prompts the user to select how features should be composed (i.e. which components of an event (guards/actions) should be merged). It highlights naming conflicts and provides a way of making the inputs disjoint before composition. This research led to the building of the prototype Feature Modelling Tool where a feature model is represented using tree-structured feature diagrams and is used to specify a product line [55]. There are some limitations of this plugin, in that, it is not possible to compose variants or theorems [54] and it does not enforce correct event fusion meaning that the user is required to ensure that the composition is performed correctly [53].

We have thus identified three distinct functionalities of this plugin: (1) composition of machines, (2) making machines disjoint before composition and (3) partial composition of machines. We can capture all of these functionalities using specification-building operators as follows.

(1) The complete composition of machines (and/or contexts) simply corresponds to the use of the `and` specification-building operator which offers a generic way of combining specifications written over both the same and different signatures. An example of this can be seen in Figure A.1 where the specification `mac3` is simply the composition of the specifications for `mac1` and `mac2` (lines 1–3). If either of these machines see a context then the context is combined in a similar fashion using the `and` operator.

```
1 spec  mac3 =
2   mac1 and mac2
3 end

4 spec  mac4 =
5   (mac1 with σ1) and
6   (mac2 with σ2)
7 end
```

```
8  spec  mac3 =
9    mac1 and (mac2 hide via σ)
10   where
11   σ : Σ_{h_mac2}  →  Σ_mac2
12   σ  =  { S_{h_mac2}  ↦  S_mac2,
13           Ω_{h_mac2}  ↦  Ω_mac2,
14           Π_{h_mac2}  ↦  Π_mac2,
15           E_{h_mac2}  =  {e1, e2} ↦ E_mac2 = {e1, e2, e3, e4},
16           V_{h_mac2}  ↦  V_mac2 }
17 end
```

Figure A.1: We use the `and` operator to completely combine features given by the machines `mac1` and `mac2` (lines 1–3) which have already been specified. We use `and` in combination with the `with` operator to ensure that the specifications of `mac1` and `mac2` are disjoint before they are merged (lines 4–7). We use the `hide via` σ operator to partially compose the features given by machines `mac1` and `mac2` (lines 8–17).

(2) We can make these specifications disjoint before composition if we wish by using the `with` specification-building operator to rename via signature morphism as shown on lines 4–7 of Figure A.1.

(3) The partial composition of machines (and/or contexts) corresponds to `and` used in conjunction with `hide via` σ, where σ pulls out the required parts of the machine(s) (or context(s)). Suppose we wish to combine MAC1 with only two of the events (e_1 and e_2 for example) from MAC2 then we can define a signature morphism as shown in Figure A.1 (lines 8–17).

In $\mathcal{EVT}$, events are composed by conjoining their guards and actions. Only events that share the same name will be composed. If there are events sharing the same name but the user does not wish for them to be combined then it is possible to rename them using the `with` specification-building operator before combining their respective machine specification. Using the specification-building operators it is possible to compose variants correctly and also ensure that proper event fusion takes place.

A.1.2 GENERIC INSTANTIATION

There are two different generic instantiation plugins, the first is by University of Southampton [108] and the second is by ETH Zurich and Hitachi. The first appears to be more mature and is the only one available directly through the "Install new Software" option in the Rodin Platform. It was last updated in 2015 and is compatible with Rodin 3.2.x. The objective is to enable the reuse of generic developments in other formal developments and to provide a mechanism for extending the instantiation to a chain of refinements.

When using the Generic Instantiation plugin, an Event-B model is referred to as a "pattern", i.e the pattern machine is the machine to be instantiated in order to refine the "problem" machine [108]. Multiple instances can be created from the same pattern to fit a specific problem. Instantiation is achieved by parametrising contexts and using the concept of a "sharing context" to allow a context be seen by several machines. The tool generates extra proof obligations to be proven to ensure that the instantiation of the pattern is a valid one and that if the instantiated machine refines an existing model that this is a valid refinement. The current problem and instance are linked via refinement and once the proof obligations have been carried out successfully then the rest of the pattern refinement chain can be instantiated.

Using this plugin, only user defined sets can be replaced so it is not possible, for example, to replace $\mathbb{Z}$, $\mathbb{N}$ or *BOOL*. Moreover, it is a requirement that all sets and contexts be replaced ensuring that there are no uninstantiated parameters. An instantiated machine and the corresponding specification-building operator representation can be seen in Figure A.2 (lines 1–11 and 12–17 respectively). We provide an illustration of this same machine rewritten using parametrisation on lines 18–27 of Figure A.2. One of the main benefits of using specification-

```
1  INSTANTIATED MACHINE maci
2    INSTANTIATES mac1  VIA ctx1
3    SEES ctx2  /* context containing
4                 the instance properties */
5    REPLACE
6      SETS s1 := s2
7      CONSTANTS c1 := c2
8    RENAME
9      VARIABLES v1 := v2
10     EVENTS e1 := e2
11 END

12 spec  maci =
13   ctx2 then
14   ((maci and ctx1)
15     with s1 ↦ s2, c1 ↦ c2,
16              v1 ↦ v2,  e1 ↦ e2
17 end
```

```
18 spec  macp[sort s1, ops c1, v1] =
19   ctx1 then
20   event e1 st =
21     ...
22 end

23 spec  maci =  =
24   ctx2 then
25   macp[mac1]
26     with s1 ↦ s2, c1 ↦ c2, v1 ↦ v2, e1 ↦ e2
27 end
```

Figure A.2: An instantiated machine obtained using the generic instantiation plugin is shown on lines 1–11. It is represented using specification-building operators on lines 12–17. Lines 18–27 use parametrisation to describe the same machine.

building operators here is that it is now possible to replace any sets that a user wishes (including non-user defined sets such as $\mathbb{Z}$, $\mathbb{N}$ or *BOOL*) once the appropriate signature morphisms have been defined by the user. The only functionality of this plugin that we cannot capture in the institution-theoretic approach is the instantiation of event parameters as these are not part of the $\mathcal{EVT}$-signature.

A.1.3 MODEL DECOMPOSITION

The Model Decomposition plugin is based on the shared variable/shared event approach that was first proposed by Abrial [5] and works for *Rodin* version 3.x [110]. To use the plugin the user selects the machine to be decomposed and defines the sub-components (machines and contexts) to be generated [109]. Then they select the style of decomposition to use (shared variable (A-style) or shared event (B-style)) and can opt

```
1 spec  M1 =
2   (M hide via  σ1)
3     with e3 ↦ e3_e
4 end

5 where σ1 = v1 ↦ v1, v2 ↦ v2, e1 ↦ e1,
6             e2 ↦ e2, e3 ↦ e3
```

```
7 spec  M2 =
8   (M hide via  σ2)
9     with e2 ↦ e2_e
10 end

11 where σ2 = v2 ↦ v2, v3 ↦ v3, e2 ↦ e2,
12             e3 ↦ e3, e4 ↦ e4
```

Figure A.3: Writing the shared variable style used in Figure 2.2 using specification-building operators

to decompose the contexts in a similar fashion. These generated sub-components can then be further refined.

The moment in development where decomposition takes place is important: decomposing early may yield an overly abstract sub-component that cannot be refined without knowledge of the others; decomposing late may mean that the already concrete model will not benefit from the decomposition. In the shared event style, if the user has the Parallel Composition plugin (which will be discussed later in Section A.1.5) installed it is possible to recompose the sub-components.

We introduced the shared variable and shared event approaches in Section 2.2.1. The shared variable method was illustrated by example in Figure 2.2 and Figure A.3 recasts this example using specification-building operators. Here, the specification-building operator `hide via` is used to hide auxiliary signature items (variables and events in this case) by means of the signature morphisms σ_1 and σ_2.

We can emulate the shared event style of decomposition in a similar fashion and the further composition of the models using the shared event (parallel) composition plugin which will be discussed in Section A.1.5.

This plugin is quite restrictive in that it is not possible to refer to the same element across sub-components. It is not possible to select which invariants are allocated to each sub-component; currently, only those

relating to variables of the sub-component are included but others can be added as theorems.

A.1.4 PATTERN

The Pattern plugin works for Rodin 3.x, it provides "design patterns" for Event-B in order to facilitate the reuse of an existing Event-B model (*pattern*) in the current Event-B model (*problem*) [47]. This allows the developer to avoid manual copying and to employ proof reuse. The plugin pattern-matches at model level and patterns are added to the Rodin database. The use case is that at some point during development the developer realises that the current model closely matches one that they have already completed as part of another project. Then they match this pattern and can incorporate it into the current model by carrying out some renamings. This allows the developer to avoid respecification and re-proof, and it also means that the refinement of the problem is generated by merging the pattern refinement with the problem.

The user must decide which guards and actions go together and these are checked for syntactic similarity. The user must also ensure that no unmatched event alters a matched variable. Both of these checks generate proof obligations which then have to be proven by the user. One of the main difficulties with using this plugin is that it requires the user to have an array of "patterns" already saved in the Rodin database. This could be resolved by providing a library of frequently used patterns for Rodin. The clone detection study that was presented in chapter 6 provides a mechanism by which a user could detect these generic patterns. The fact that the refinement of the pattern is also included can be advantageous, but this is unnecessary if the pattern is only matched in an abstract machine and the refinement proceeds in a different way. In the chapter 7 we identified the detection and analysis of clone genealogies

```
1 spec  mac1 =
2   pattern with σ
3   then
4   ...
5 end
```

Figure A.4: Incorporating a `pattern` machine in the current development `mac1`

as future work, and this could also be used to investigate this plugin's approach in greater detail.

Figure A.4 illustrates, using specification-building operators, how to utilise a pattern machine specification (`pattern`). Here `with` σ denotes the renamings (via signature morphism) to be carried out. The specification-building operators do not directly facilitate pattern matching; however, they offer a mechanism by which an already completed specification can be included in a current one. They also provide the advantage of not requiring the user to decide which guards and actions go together because events with the same name are automatically fused. Our clone detector could potentially be reused to match patterns with their clones in this way.

A.1.5 SHARED EVENT (PARALLEL) COMPOSITION

This plugin goes by the name "Parallel Composition" in the literature and "Shared Event Composition" on the Rodin wiki. The composition is based on that proposed by Butler [17]. It uses CSP-style parallel composition $\|$ to compose machines through events. This is the precursor to event refinement structures (ERS) which are sometimes referred to as atomicity decomposition [43]. ERS are used to automatically generate Event-B from control-flow and refinement diagrams. This plugin works with *Rodin* 3.x. and its main limitation is that there are no facilities for composing variants or theorems.

```
1 spec  mac3 =
2   (mac1 with σ1) and (mac1 with σ2)
3   where
4     σ1  =  {e1  ↦  e3}
5     σ2  =  {e2  ↦  e3}
```

Figure A.5: Shared Event Composition using the specification-building operators

In order to compose the events of a machine (presentation) in $\mathcal{EVT}$ we simply rename the events to be composed so that they share the same name and then include all machines to be composed into the same specification. The CASL notion of "same name, same thing" means that any like-named events will be fused by default. Figure A.5 shows how to compose the machines `mac1` and `mac2` by merging the events `e1` in `mac1` and `e2` in `mac2`. By renaming both of the events to `e3` we merge them in the combined machine `mac3`. Of course, one must be careful to ensure that there are no other events in the final machine called `e3` that are not intended to be merged. This approach also enables us to compose variants.

A.1.6 MODULARISATION PLUGIN

The modularisation plugin was inspired by the modularisation approach to decomposition in Event-B which was outlined in Section 2.2.1. Here, *modules* split up an Event-B component (machine/context) and are paired with an interface defining the conditions for incorporating one module into another [67]. *Module interfaces* are a new type of Event-B component that list the operations contained in the module. These are similar to machines but they may not specify events. The events of a machine which imports an interface can see the visible constants, sets and axioms, call the imported operations, and the interface variables and invariants are added to the machine. The imported interface variables can be referred

to by invariants, guards and actions but may not be directly updated by an action. Although similar to the shared variable approach this method is less restrictive, as invariants can be included in the module interface.

This plugin provides a mechanism for modularity in Event-B but there are a large number of different features that the user needs to utilise and it is unclear how a model developed using these constructs might be translated into/combined with a different formalism (or even with a different plugin).

This plugin is a special case of shared variable decomposition which is expressed using the specification-building operators in Figure A.3. The plugin provides further structures such as interfaces and modules but, by using the specification-building operators, these extra features are not necessary to achieve this kind of modularisation.

A.1.7 XEVENT-B

This is a new plugin that was released for Rodin in 2017. It provides text editors for Event-B contexts and machines. With regards to modularisation, it provides a way of combining machines called *machine inclusion*[1]. Figure A.6 illustrates how the plugin text editors look accompanied by a version of the same specification written using the specification-building operators. In fact, this particular example (cars on a bridge taken from [3]) is fully described in Appendix B using specification-building operators in $\mathcal{EVT}$. From the example shown in Figure A.6, it is clear that the synchronises keyword corresponds to the application of a signature morphism to the event names, i.e.

`ML_out.Initialisation` $\mapsto$ `Car_m1_SNSR.Initialisation`

1 http://wiki.event-b.org/index.php/XEvent-B

```
1 MACHINE Car_m1_SNSR
2   includes  Sensor_m0_SNSR  as  ML_out IL_out
3   ...
4   EVENTS
5     Initialisation
6        synchronises  ML_out.INITIALISATION
7        synchronises  IL_out.INITIALISATION
8   ...
9 END
```

```
1 spec  Car_m1_SNSR =
2   (Sensor_m0_SNSR with σ1)
3   and (Sensor_m0_SNSR with σ2)
4   then
5     ...
6 end
```

Figure A.6: Representing the functionality of the XEvent-B plugin using specification-building operators.

A.1.8 RELATED PLUGINS

We have identified a further two Rodin plugins that are relevant to this discussion but that do not fall directly under the heading of modularisation plugins. These are the Theory Extension and Rename Refactoring plugins.

1. *Theory Extension*

This plugin facilitates the addition of new data types, operators and axioms ("theories") for use in multiple independent Event-B developments. This is easily achieved in $\mathcal{EVT}$ or even by specifying a new data type in $\mathcal{FOPEQ}$ and using the `and` or `then` specification-building operator (with the comorphism from $\mathcal{FOPEQ}$ to $\mathcal{EVT}$ applied whenever necessary) to make this new data-type available for use in other developments.

2. *Rename Refactory*

This Rename Refactory plugin also goes by the name "Refactoring Framework"[2]. The Rename Refactory plugin works on *Rodin* 3.x and facilitates the renaming of variables, parameters, carrier sets, constants, events and other labelled elements (invariants, axioms, guards, etc).

2 http://wiki.event-b.org/index.php/Refactoring_Framework

This plugin pushes renamings through to the proof obligation level so that proofs do not need to be redone. The last modification to this plugin was in 2014. Although this particular plugin is very good at renaming, it doesn't offer much in terms of increasing the modularity of the Event-B formalism. We included in it our discussion here because it offers a very useful feature for Event-B users and our objective is to show that its functionality can also be captured using specification-building operators.

It is easy to see that the functionality of this plugin can be expressed by using the `with` specification-building operator which facilitates the application of a signature morphism to rename signature items (sorts represented as carrier sets, operations, predicates, variables and events). However, `with` will not enable us to rename event parameters or labels as these are not part of an $\mathcal{EVT}$ signature.

A.1.9 RODIN COMPATABILITIES

The *Rodin Platform* version compatibility of each of these plugins is summarised in Table A.1. The majority of this data was obtained from the *Rodin* wiki[3]. In any case where this information for a particular plugin was not available here we looked to the literature. From this table it is clear to see that most of these plugins work with recent versions of *Rodin*. The Feature Composition, Generic Instantiation and Modularisation plugins are the only ones not explicitly available for *Rodin* 3.x.x at the time of writing.

3 http://wiki.event-b.org/index.php/Rodin_Platform_3.2.0_External_Plug-ins

Plugin Name	***Rodin* Version Compatibility**
Feature Composition	2.0
Generic Instantiation (Soton)	3.2.x
Model Decomposition	3.x.x
Pattern	3.x.x
Shared Event Composition	3.x.x
Modularisation Plugin	2.x.x
XEvent-B	3.3
Theory Extension	3.1.x
Rename Refactoring	3.x.x

Table A.1: Table summarising the latest version of the *Rodin Platform* that each of the identified plugins is compatible with.

A.2 REFINEMENT AS A MODULARISATION TECHNIQUE

Refinement bears some similarity to the "divide and conquer" approach advocated by modularisation in the sense that the system is divided into simpler pieces that will be incrementally added to each other in further refinement steps. However, the specification remains monolithic and this approach does not facilitate the development of a system by developers working in parallel which is a major advantage of modularity in software engineering.

Refinement in Event-B facilitates the carrying of proofs from the abstract machine in order to assist the provers in proving more complex proof obligations which may be encountered in the concrete machine [6]. As described in Section 2.1.1, there are two primary types of refinement in Event-B: data refinement and superposition refinement. During data refinement, the user must supply *gluing invariant* to relate the prop-

erties of the abstract machine to their counterparts in the concrete machine. Any such gluing invariant must be true upon `Initialisation` of both machines and be preserved by all events. A comprehensive behavioural semantics for refinement in Event-B in terms of CSP refinement can be found in [106]. Superposition refinement occurs when new sentences are added to the machine in such a way that they are superimposed onto their abstract components. This amounts to adding new events, invariants or new sentences to events such as guards or actions. Rodin generates a number of refinement-specific proof obligations as described in Section 2.1.2 which ensure that the refinement steps taken are valid ones.

Refinement in Institution Theory is captured using model-class inclusion as discussed in Chapter 3. The central rule being that the model-class of a concrete specification, C, must be a subset of the model-class of its corresponding abstract specification, A [100]. In this way, the concrete specification should only exhibit behaviours that were possible in the abstract case (provided that the signatures of C and A are the same). In the case where the signatures are different, we apply this check for model-class inclusion alongside the model reduct. The model reduct is used to restrict the model-class of the concrete specification to only those models that appear in the model-class of the abstract specification.

Figure A.7 illustrates how institution-theoretic parametrisation can be used to represent data refinement between the abstract and refined traffic light example that was written in $\mathcal{EVT}$ in Figures 3.5 and 3.6. In Event-B, data refinement was achieved using gluing invariants, here the `hide via` specification-building operator alongside parametrisation corresponds to these gluing invariants.

LINES 1–7: The specification AbsLight defines a single abstract traffic light, it takes a specification over the same signature as `TwoBools` (originally defined in Chapter 3) as a parameter (between the square

brackets on line 1). Two events are added that can refer to elements of the parameter specification given (`TwoBools`). It is possible to write either a specification name or a full specification as a parameter. As this is the definition of `AbsLight` we call the parameter specification on line 1 the *formal parameter*. When we instantiate it later we will supply an *actual parameter*. The axioms (predicates) of the actual parameter must imply those of the formal parameter that was supplied when the specification was defined.

LINES 8–14: This is a specification for a traffic light system with two lights. Notice that on lines 9 and 12 we provide the *actual* parameter for the specification for each instantiation of `AbsLight` and these must have the same signature as that of the *formal* parameter specification on line 1. A parameter that does not meet this requirement is not valid [100].

LINES 15–18: This is the specification of a refined light that uses colours instead of boolean flags. Line 16 takes a parametrised `AbsLight` specification with the specification `TwoColours` as a parameter. The `hide via` specification-building operator essentially describes the refinement relation between `TwoColours` and `TwoBools` by including the signature morphism between them. In this way we can treat `TwoColours` as a specification over same signature as `TwoBools` so it can be passed as a valid parameter in order to instantiate a copy of `AbsLight`. Line 17 uses `with` to rename the events appropriately.

LINES 19–23: This is the `TwoColours` specification.

With regard to superposition refinement, the $\mathcal{CASL}$ notion of "same name, same thing" ensures that all like-named events are merged thus supplying a means for achieving superposition refinement in the institutional setting.

```
1 spec  AbsLight[sort Bool, ops i_go, u_go : Bool, . i_go ≠ u_go] =
2     event go =
3       when u_go = false
4       thenAct i_go = true
5     event stop =
6       thenAct i_go = false
7 end

8 spec  AbsTwoLights =
9   AbsLight[TwoBools]
10    with i_go ↦ peds_go, u_go ↦ cars_go, go ↦ set_pedsgo, stop ↦ set_pedsstop
11  and
12  AbsLight[TwoBools]
13    with i_go ↦ cars_go, u_go ↦ peds_go, go ↦ set_carsgo, stop ↦ set_carsstop
14 end

15 spec  Reflight =
16   AbsLight[TwoColours hide via Bool ↦ Colour, true ↦ green, false ↦ red,
17           i_go ↦ i_col, u_go ↦ u_col]
18   with go ↦ set_green, stop ↦ set_red
19 end

20 spec  TwoColours =
21   sort Colour
22   ops i_col, u_col, green, red:Colour
23   . i_col ≠ u_col
24 end
```

Figure A.7: A parametrised version of the simple traffic light system that was illustrated in Figure 2.1.

A.3 DISCUSSION

The identified approaches and plugins offer improvements to the Rodin Platform in order to facilitate modular development in Event-B although none of them directly enhance the Event-B formal specification language itself. The use of specification-building operators and parametrisation from the theory of institutions offers a more mathematically grounded and uniform approach to increasing the modularity of the Event-B formal specification language. This approach is facilitated through the use of $\mathcal{EVT}$ which is our institution for Event-B [40, 41]. The main contribution of this appendix is that we have shown that all of the plugins described in Table 2.1 can be expressed (and some even improved) using these techniques. Thus, the theory of institutions and our definition

of $\mathcal{EVT}$ has resulted in a more generic approach to modularisation in Event-B.

We have identified refinement as a modularisation technique in Event-B, although it is generally viewed as a program development technique in other programming paradigms, and shown how it can be captured using institution-theoretic parametrisation and the specification-building operators [48]. A language for refinement, which is an extension to $\mathcal{CASL}$ and is based on the concept of constructors, has been introduced and we intend to apply it to the $\mathcal{EVT}$-specifications that are presented throughout this thesis as future work [23].

B

MATHEMATICAL NOTATION AND SOFTWARE ARTEFACTS

In this appendix, we briefly summarise the mathematical notation that we have used throughout this thesis. We also describe the software artefacts that were constructed as part of this project.

B.1 MATHEMATICAL NOTATION

The Event-B mathematical language at the base of the three-layer model illustrated in Figure 4.1 is set theory. We use this set-theoretic language throughout this thesis and we have summarised the symbols that we have used in Table B.1.

B.2 INSTITUTIONS

The institutions referenced throughout this thesis are summarised below with the page number of their definition.

$\mathcal{FOPEQ}$ The Institution for First-Order Predicate Logic with Equality 28

$\mathcal{EVT}$ The Institution for Event-B built on $\mathcal{FOPEQ}$ 41

$\mathcal{ACT}$ The Institution for Actions 104

$\mathcal{UML}$ The Institution for UML State Machines 103

Symbol	Description
$\mapsto$	The maplet symbol is used to represent an ordered pair.
$\{x \mid P(x)\}$	Set comprehension : For some predicate P, this is the set of all x such that $P(x)$ holds.
$\lhd$	Domain restriction operator.
$⩤$	Domain anti-restriction operator.
$\sqsubseteq$	Refinement.
dom	Function to access the domain of a function.
$\wp$	Power Set.

Table B.1: Summary of the mathematical notation used throughout this thesis.

B.3 INSTITUTION COMORPHISMS AND SEMI-MORPHISMS

The institution comorphisms and semi-morphisms referenced throughout this thesis are summarised below with the page number of their definition.

$\rho : \mathcal{FOPEQ} \to \mathcal{EVT}$ Comorphism between $\mathcal{FOPEQ}$ and $\mathcal{EVT}$ 60

$\rho : \mathcal{UML} \to \mathcal{EVT}$ Comorphism between $\mathcal{UML}$ and $\mathcal{EVT}$ 114

$\rho_{BASE} : \mathcal{FOPEQ} \to \mathcal{ACT}$ Comorphism between $\mathcal{FOPEQ}$ and $\mathcal{ACT}$ 113

B.4 SOFTWARE ARTEFACTS

The software artefacts that resulted from this thesis can be found on our github page at `https://github.com/mariefarrell/phdartefacts.git`. This repository is comprised of a number of folders and we summarise their contents below.

B.4.1 CLONEDETECTOR

The clone detector repository contains the suite of Python3 programs that we used to analyse our corpus of Event-B specifications in Chapter 6. In what follows, assume that the directory you want to process is called `'DIR'`.

`eventb.py`: contains the data structures for Event B machines, contexts events etc. It is not runnable.

`bumparser.py`: parses a .bum/.buc file and represents it using the eventb classes. It is run by typing:

```
python3 bumparser.py DIR
```

`metrics.py`: reads in the .bum/.buc files and prints out their metrics (one line per context/machine/event). It is run by typing:

```
python3 metrics.py DIR
```

`tokeniser.py`: tokenises the sentences (guards, actions etc.) and can also do anonymisation. It is run by typing:

```
python3 tokeniser.py -anon DIR
```

The `-anon` switch anonymises the variable names. The `-inv`, `-init` and `-context`/`-event` switches can be used as in lcs_compare below.

`lcs_compare.py`: is the non-commutative clone detector. It is run by typing:

```
python3 lcs_compare DIR
```

By default, this groups on machines and the optional switches are:

`-event`: Group on events.

`-context`: Group on contexts.

- `-init` Include `Initialisation` events. By default we exclude these as all `Initialisation` events assign variables to values and thus are all clones of each other to some degree.
- `-inv` Include invariants and variants. At the machine level this includes the machine invariants and variants, whereas at event level this includes the all machine invariants and the variant is only included for non-`ordinary` events.

`micropatterns.py`: looks for micropatterns i.e. complete macines/contexts/events (with > 2 axiom/action sentences) that occur multiple times. It can be run by typing:

```
python3 micropatterns.py -anon -event DIR
```

B.4.2 EB2EVT

The `EB2EVT` translational semantics parser consists of four Haskell programs.

`FOPEQ.hs`: corresponds to the $\mathcal{FOPEQ}$ interface in Figure 4.3.

`Syntax.hs`: contains the abstract data types for the various components of an Event-B specification. This corresponds to the Event-B syntax shown in Figure 4.2.

`Semantics.hs`: encodes the semantic functions as described in Figures 4.7, 4.8, 4.9 and 4.10.

`ParseEb.hs`: parses the Event-B specification and generates the corresponding $\mathcal{EVT}$-specifications. This can be run by typing the following command into `ghci`, once the file has been loaded:

```
parseDirectory "DIR"
```

B.4.3 EVTHETS

This folder contains the HETS prototype of $\mathcal{EVT}$. A number of utility files are included such as `Keywords.hs`, `ParseEVT.hs`, `StaticAnalysis.hs`, `SymbolParser.hs`, `ATC.der.hs` and `AS.der.hs`. These contain abstract data types describing the abstract syntax of $\mathcal{EVT}$-specifications and provisions for parsing and statically analysing these specifications. $\mathcal{EVT}$-signatures and signature morphisms are defined in `SignEVT.hs`. In HETS, every institution is represented as a 'logic' (`Logic.hs`) and comorphisms can be defined between them (`Comorphisms.hs`). We have utilised the HETS implementation of the $\mathcal{CASL}$ institution for the $\mathcal{FOPEQ}$ components of our implementation of $\mathcal{EVT}$.

B.4.4 SPECS

This folder contains a set of modular $\mathcal{EVT}$-specifications that were used to investigate how the specification-building operators can be applied to $\mathcal{EVT}$-specifications.

BIBLIOGRAPHY

[1] *About the Unified Modeling Language Specification Version 2.5*. URL: http://www.omg.org/spec/UML/2.5/ (visited on 10/27/2017).

[2] Jean-Raymond Abrial. *Event model decomposition*. Tech. rep. Department of Computer Science, ETH Zurich, 2009.

[3] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. 1st. Cambridge University Press, 2010.

[4] Jean-Raymond Abrial. "From Z to B and then Event-B: Assigning Proofs to Meaningful Programs". In: *10th International Conference on Integrated Formal Methods, IFM*. Vol. 7940. LNCS. 2013, pp. 1–15.

[5] Jean-Raymond Abrial and Stefan Hallerstede. "Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B". In: *Fundamenta Informaticae* 77.1-2 (2007), pp. 1–28.

[6] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. "Rodin: an open toolset for modelling and reasoning in Event-B". In: *International Journal on Software Tools for Technology Transfer* 12.6 (2010), pp. 447–466.

[7] Grigoris Antoniou and Frank Van Harmelen. "Web Ontology Language: OWL". In: *Handbook on ontologies*. Springer, 2004, pp. 67–92.

[8] Ralph-Johan Back. "Refinement calculus, part II: Parallel and reactive programs". In: *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*. Vol. 430. LNCS. 1990, pp. 67–93.

[9] Ralph-Johan Back and Kaisa Sere. "Stepwise refinement of action systems". In: *International Conference on Mathematics of Program Construction, MPC*. Vol. 375. LNCS. 1989, pp. 115–138.

[10] Ralph-Johan Back and Joakim von Wright. "Refinement calculus, part I: Sequential nondeterministic programs". In: *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*. Vol. 430. LNCS. 1989, pp. 42–66.

[11] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A systematic Introduction*. Springer, 1998.

[12] Michael Barnett, K. Rustan M. Leino, and Wolfram Schulte. "The Spec# programming system: An overview". In: *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, CASSIS*. Vol. 3362. LNCS. 2004, pp. 49–69.

[13] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. "Boogie: A modular reusable verifier for object-oriented programs". In: *4th International Symposium on Formal Methods for Components and Objects, FMCO*. Vol. 4111. LNCS. 2005, pp. 364–387.

[14] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. "Clone detection using abstract syntax trees". In: *14th International Conference on Software Maintenance, ICSM*. IEEE. 1998, pp. 368–377.

[15] Rod M. Burstall and Joseph A. Goguen. "Putting theories together to make specifications". In: *5th International Joint Conference on Artificial intelligence, IJCAI*. Vol. 2. Morgan Kaufmann. 1977, pp. 1045–1058.

[16] Rod M. Burstall and Joseph A. Goguen. "The semantics of Clear, a specification language". In: *Abstract Software Specifications*. Vol. 86. LNCS. 1980, pp. 292–332.

[17] Michael Butler. "Decomposition Structures for Event-B". In: *7th International Conference on Integrated Formal Methods, IFM*. Vol. 5423. LNCS. 2009, pp. 20–38.

[18] Michael Butler and Issam Maamria. "Practical theory extension in Event-B". In: *Theories of Programming and Formal Methods*. Vol. 8051. LNCS. 2013, pp. 67–81.

[19] Néstor Catano, K. Rustan M. Leino, and Víctor Rivera. "The EventB2Dafny Rodin plug-in". In: *2nd Workshop on Developing Tools as Plug-ins, TOPI*. IEEE. 2012, pp. 49–54.

[20] Néstor Catano and Víctor Rivera. "EventB2Java: A code generator for Event-B". In: *8th International NASA Formal Methods Symposium, NFM*. Vol. 9690. LNCS. 2016, pp. 166–171.

[21] Néstor Cataño, Tim Wahls, Camilo Rueda, Víctor Rivera, and Danni Yu. "Translating B machines to JML specifications". In: *27th ACM Symposium on Applied Computing, SAC*. ACM. 2012, pp. 1271–1277.

[22] Robert N Charette. "Why software fails". In: *IEEE spectrum* 42.9 (2005), p. 36.

[23] Mihai Codescu, Till Mossakowski, Donald Sannella, and Andrzej Tarlecki. "Specification refinements: Calculi, tools, and applications". In: *Science of Computer Programming* 144.Supplement C (2017), pp. 1 –49.

[24] Kriangsak Damchoom. "An incremental refinement approach to a development of a flash-based file system in Event-B". PhD thesis. University of Southampton, 2010.

[25] Kriangsak Damchoom, Michael Butler, and Jean-Raymond Abrial. "Modelling and proof of a tree-structured file system in Event-B and Rodin". In: *10th International Conference on Formal Engineering Methods, ICFEM 2008*. Vol. 5256. LNCS. 2008, pp. 25–44.

[26] Răzvan Diaconescu. “Grothendieck institutions”. In: *Applied Categorical Structures* 10.4 (2002), pp. 383–402.

[27] Răzvan Diaconescu. *Institution-Independent Model Theory*. Springer, 2008.

[28] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ report: The language, proof techniques, and methodologies for object-oriented algebraic specification*. Vol. 6. World Scientific, 1998.

[29] Răzvan Diaconescu and Kokichi Futatsugi. “Logical foundations of CafeOBJ”. In: *Theoretical Computer Science* 285.2 (2002), pp. 289–318.

[30] Edsger W. Dijkstra. “Guarded commands, nondeterminacy and formal derivation of programs”. In: *Communications of the ACM* 18.8 (1975), pp. 453–457.

[31] Edsger W. Dijkstra. *A discipline of programming*. Prentice-Hall, 1976.

[32] Ionut Dinca, Florentin Ipate, Laurentiu Mierla, and Alin Stefanescu. “Learn and test for Event-B–a Rodin plugin”. In: *3rd International Conference on Abstract State Machines, Alloy, B, VDM, and Z, ABZ*. Vol. 7316. LNCS. 2012, pp. 361–364.

[33] Andrew Edmunds and Michael Butler. “Tasking Event-B: An extension to Event-B for generating concurrent code”. In: *PLACES*. 2011.

[34] Hartmut Ehrig, Bernd Mahr, Ingo Classen, and Fernando Orejas. “Introduction to algebraic specification. Part 1: Formal methods for software development”. In: *The Computer Journal* 35.5 (1992), pp. 460–467.

[35] Manuel Fähndrich. “Static Verification for Code Contracts.” In: *17th International Symposium on Static Analysis, SAS*. Vol. 6337. LNCS. 2010, pp. 2–5.

[36] Marie Farrell. “Using the Theory of Institutions to Integrate Software Models via Refinement”. In: *PhD Symposium at the 12th International Conference on Integrated Formal Methods,PhD-iFM*. 2016.

[37] Marie Farrell, Rosemary Monahan, and James F. Power. “An Approach to Integrating Software Models via Refinement (Poster)”. In: *ACM womENcourage*. 2014.

[38] Marie Farrell, Rosemary Monahan, and James F. Power. “A Logical Framework for Integrating Software Models via Refinement”. In: *British Colloquium for Theoretical Computer Science, BCTCS*. 2016.

[39] Marie Farrell, Rosemary Monahan, and James F. Power. “Modularising and Promoting Interoperability for Event-B Specifications using Institution Theory (Poster)”. In: *28th European Summer School in Logic, Language and Information, ESSLLI*. 2016, p. 74.

[40] Marie Farrell, Rosemary Monahan, and James F. Power. “Providing a Semantics and Modularisation Constructs for Event-B using Institutions”. In: *23rd International Workshop on Algebraic Development Techniques, WADT*. 2016, pp. 18–19.

[41] Marie Farrell, Rosemary Monahan, and James F. Power. “An Institution for Event-B”. In: *Recent Trends in Algebraic Development Techniques. WADT 2016.* Vol. 10644. LNCS. 2017, pp. 104–119.

[42] Marie Farrell, Rosemary Monahan, and James F. Power. “Specification Clones: An empirical study of the structure of Event-B specifications”. In: *15th International Conference on Software Engineering and Formal Methods, SEFM*. Vol. 10469. LNCS. 2017, pp. 152–167.

[43] Asieh Salehi Fathabadi, Michael Butler, and Abdolbaghi Rezazadeh. “Language and tool support for event refinement structures in Event-B”. In: *Formal Aspects of Computing* 27.3 (2015), pp. 499–523.

[44] Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. "Isabelle/Circus: A Process Specification and Verification Environment." In: *4th International Conference on Verified Software: Theories, Tools, Experiments , VSTTE*. Vol. 7152. LNCS. 2012, pp. 243–260.

[45] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3—where programs meet provers". In: *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP*. Vol. 7792. LNCS. 2013, pp. 125–128.

[46] John Fitzgerald, Peter Gorm Larsen, and Jim Woodcock. "Foundations for model-based engineering of systems of systems". In: *Complex Systems Design & Management*. Springer, 2014, pp. 1–19.

[47] Andreas Fürst. "Design patterns in Event-B and their tool support". MA thesis. Department of Computer Science, ETH Zurich, 2009.

[48] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, 2002.

[49] Joseph A. Goguen. "A categorical manifesto". In: *Mathematical Structures in Computer Science* 1.1 (1991), pp. 49–67.

[50] Joseph A. Goguen and Rod M. Burstall. "Introducing institutions". In: *Workshop on Logic of Programs*. Vol. 164. LNCS. 1983, pp. 221–256.

[51] Joseph A. Goguen and Rod M. Burstall. "A study in the foundations of programming methodology: Specifications, institutions, charters and parchments". In: *Category Theory and Computer Programming*. Vol. 240. LNCS. 1986, pp. 313–333.

[52] Joseph A. Goguen and Rod M. Burstall. "Institutions: abstract model theory for specification and programming". In: *Journal of the ACM* 39.1 (1992), pp. 95–146.

[53] Ali Gondal, Michael Poppleton, and Michael Butler. "Composing Event-B Specifications-Case-Study Experience". In: *10th International Conference on Software Composition, SC*. Vol. 6708. LNCS. 2011, pp. 100–115.

[54] Ali Gondal, Michael Poppleton, and Colin Snook. "Feature composition-towards product lines of Event-B models". In: *3rd International Workshop on Model-Driven Product Line Engineering, MDPLE*. Vol. 6698. LNCS. 2009, pp. 18–25.

[55] Ali Gondal, Michael Poppleton, Michael Butler, and Colin Snook. "Feature-Oriented Modelling Using Event-B". In: *International Conference on Software Engineering Theory and Practice, SETP*. ISRST, 2010, pp. 100–106.

[56] Jason Gross, Adam Chlipala, and David I Spivak. "Experience implementing a performant category-theory library in Coq". In: *5th International Conference on Interactive Theorem Proving*. Vol. 8558. LNCS. 2014, pp. 275–291.

[57] Martin Große-Rhode. *Semantic Integration of Heterogeneous Software Specifications*. Springer, 2013.

[58] Stefan Hallerstede. "Justifications for the Event-B modelling notation". In: *7th International Conference of B Users*. Vol. 4355. LNCS. 2007, pp. 49–63.

[59] Stefan Hallerstede. "On the purpose of Event-B proof obligations". In: *1st International Conference on Abstract State Machines, B and Z, ABZ*. Vol. 5238. LNCS. 2008, pp. 125–138.

[60] Thai Son Hoang, Alexei Iliasov, Renato Silva, and Wei Wei. "A survey on Event-B decomposition". In: *Electronic Communications of the EASST* 46 (2011), pp. 1–15.

[61] Thai Son Hoang, Colin Snook, Lukas Ladenberger, and Michael Butler. "Validating the Requirements and Design of a Hemodialysis Machine Using iUML-B, BMotion Studio, and Co-Simulation". In: *5th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Vol. 9675. LNCS. 2016, pp. 360–375.

[62] Charles Anthony Richard Hoare. "Unified Theories of Programming". In: *Mathematical Methods in Program Development*. Vol. 158. NATO ASI Series (Series F: Computer and Systems Sciences). 1997, pp. 313–367.

[63] Charles Antony Richard Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[64] Charles Antony Richard Hoare. "Communicating sequential processes". In: *The Origin of Concurrent Programming*. Springer, 1978, pp. 413–443.

[65] Gérard Huet and Gordon Plotkin. *Logical frameworks*. Cambridge University Press, 1991.

[66] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[67] Alexei Iliasov, Elena Troubitsyna, Linas Laibinis, Alexander Romanovsky, Kimmo Varpaaniemi, Dubravka Ilic, and Timo Latvala. "Supporting Reuse in Event-B Development: Modularisation Approach". In: *2nd International Conference on Abstract State Machines, Alloy, B and Z, ABZ*. Vol. 5977. LNCS. 2010, pp. 174–188.

[68] Cliff B. Jones. *Systematic software development using VDM*. Vol. 2. Prentice-Hall, 1990.

[69] Yonit Kesten and Amir Pnueli. "Modularization and abstraction: The keys to practical formal verification". In: *40th International Symposium on Mathematical Foundations of Computer Science, MFCS*. Vol. 9234. LNCS. 1998, pp. 54–71.

[70] Tibor Kiss and Katalin Tünde Jánosi-Rancz. "Developing railway interlocking systems with session types and Event-B". In: *11th International Symposium on Applied Computational Intelligence and Informatics, SACI*. IEEE. 2016, pp. 93–98.

[71] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. "Robust statistical methods for empirical software engineering". In: *Empirical Software Engineering* 22 (2 2017), pp. 579–630.

[72] Anneke G. Kleppe, Jos B. Warmer, and Wim Bast. *MDA Explained: The model driven architecture: practice and promise*. Addison-Wesley, 2003.

[73] Alexander Knapp, Till Mossakowski, Markus Roggenbach, and Martin Glauer. "An Institution for Simple UML State Machines". In: *18th International Conference on Fundamental Approaches to Software Engineering, FASE*. Vol. 9033. LNCS. 2015, pp. 3–18.

[74] Ch. Suresh Kumar, D. Raghu, and P. Ratna Kumar. "A Domestic Case Studies Probability to Overcome Software Failures". In: *Journal of Telematics and Informatics* 1.1 (2013), pp. 20–25.

[75] Linas Laibinis, Elena Troubitsyna, Alexei Iliasov, and Romanovsky Alexander. "Formal Development of the BepiColombo Pilot". In: DEPLOY Planery Meeting, November 2008, Turku.

[76] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[77] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. "Preliminary Design of JML: A Behavioural Interface Specification Language for Java". In: *ACM SIGSOFT Software Engineering Notes* 31.3 (2006), pp. 1–38.

[78] K. Rustan M. Leino. "Dafny: An automatic program verifier for functional correctness". In: *16th International Conference on Logic for Programming Artificial Intelligence and Reasoning, LPAR*. Vol. 6355. LNCS. 2010, pp. 348–370.

[79] K. Rustan M. Leino, Peter Müller, and Jan Smans. "Verification of Concurrent Programs with Chalice." In: *Foundations of Security Analysis and Design, FOSAD*. Vol. 5705. LNCS. 2009, pp. 195–222.

[80] Atif Mashkoor. "The hemodialysis machine case study". In: *5th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Vol. 9675. 2016, pp. 329–343.

[81] David Mentré, Claude Marché, Jean-Christophe Filliâtre, and Masashi Asuka. "Discharging proof obligations from Atelier B using multiple automated provers". In: *3rd International Conference on Abstract State Machines, Alloy, B, VDM, and Z, ABZ*. Vol. 7316. LNCS. 2012, pp. 238–251.

[82] Dominique Méry and Neeraj Kumar Singh. "Automatic Code Generation from event-B Models". In: *2nd Symposium on Information and Communication Technology, SoICT*. ACM, 2011, pp. 179–188.

[83] Dominique Méry and Neeraj Kumar Singh. "Functional behavior of a cardiac pacing system". In: *International Journal of Discrete Event Control Systems* 1.2 (2011), pp. 129–149.

[84] José Meseguer. "General logics". In: *Studies in Logic and the Foundations of Mathematics* 129 (1989), pp. 275–329.

[85] Cornelis Adam Middelburg. "VVSL: A language for structured VDM specifications". In: *Formal Aspects of Computing* 1 (1989), pp. 115–135.

[86] Carroll Morgan, Ken Robinson, and Paul Gardiner. *On the Refinement Calculus*. Springer, 1988.

[87] Joseph Morris. "A Theoretical Basis for Stepwise Refinement and the Programming Calculus". In: *Science of Computer Programming* 9.3 (1987), pp. 287–306.

[88] Till Mossakowski, Răzvan Diaconescu, and Andrzej Tarlecki. "What is a logic translation?" In: *Logica Universalis* 3.1 (2009), pp. 95–124.

[89] Till Mossakowski, Christian Maeder, and Klaus Lüttich. "The Heterogeneous Tool Set, HETS". In: *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*. Vol. 4424. LNCS. 2007, pp. 519–522.

[90] Peter D. Mosses, ed. *CASL Reference Manual*. Vol. 2960. LNCS. 2004.

[91] Liam O'Reilly. "Structured Specification with Processes and Data". PhD thesis. Department of Computer Science, Swansea University, 2012.

[92] Marta Olszewska and Kaisa Sere. "Specification Metrics for Event-B Developments". In: *International Conference on Quality Engineering in Software Technology, ICSQ*. 2010.

[93] Sergey Ostroumov and Leonidas Tsiopoulos. "VHDL code generation from formal Event-B models". In: *14th Euromicro Conference on Digital System Design, DSD*. IEEE. 2011, pp. 127–134.

[94] M. Pitu, D. Grijincu, P. Li, A. Saleem, Rosemary Monahan, and Diarmuid P. O'Donoghue. "Arís: Analogical Reasoning for reuse of Implementation & Specification." In: *4th International Workshop on Artificial Intelligence for Formal Methods, AI4FM, HW-MACS-TR-0100*. 2013, pp. 13–16.

[95] Michael Poppleton. "The composition of Event-B models". In: *1st International Conference on Abstract State Machines, B and Z, ABZ*. Vol. 5238. LNCS. 2008, pp. 209–222.

[96] Steve Reeves and David Streader. "General Refinement, Part One: Interfaces, Determinism and Special Refinement". In: *Electronic Notes in Theoretical Computer Science* 214 (2008), pp. 277–307.

[97] Steve Reeves and David Streader. "General Refinement, Part two: Flexible Refinement". In: *Electronic Notes in Theoretical Computer Science* 214 (2008), pp. 309–329.

[98] Chanchal K. Roy, Minhaz F. Zibran, and Rainer Koschke. "The vision of software clone management: past, present, and future". In: *Software Maintenance, Reengineering and Reverse Engineering*. IEEE, 2014, pp. 18–33.

[99] David E. Rydeheard and Rod M. Burstall. *Computational Category Theory*. Vol. 152. Prentice-Hall, 1988.

[100] Donald Sanella and Andezej Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Springer, 2012.

[101] Donald Sannella and Andrzej Tarlecki. "Specifications in an arbitrary institution". In: *Information and computation* 76.2-3 (1988), pp. 165–210.

[102] Matthias Schmalz. *The Logic of Event-B*. Technical Report 698. Department of Computer Science, ETH Zurich, 2010.

[103] Matthias Schmalz. "Term Rewriting in Logics of Partial Functions." In: *19th International Conference on Formal Engineering Methods, ICFEM*. Vol. 6991. LNCS. 2011, pp. 633–650.

[104] Steve Schneider. *The B-method: An introduction*. Palgrave, 2001.

[105] Steve Schneider, Helen Treharne, and Heike Wehrheim. "A CSP approach to control in Event-B". In: *8th International Conference on Integrated Formal Methods, IFM*. Vol. 6396. LNCS. 2010, pp. 260–274.

[106] Steve Schneider, Helen Treharne, and Heike Wehrheim. "The behavioural semantics of Event-B refinement". In: *Formal Aspects of Computing* 26 (2014), pp. 251–280.

[107] Dana S. Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*. Vol. 1. Technical Monograph PRG-6, Oxford University Computing Laboratory, Programming Research Group, 1971.

[108] Renato Silva and Michael Butler. "Supporting reuse of Event-B developments through generic instantiation". In: *11th International Conference on Formal Engineering Methods, ICFEM*. Vol. 5885. LNCS. 2009, pp. 466–484.

[109] Renato Silva and Michael Butler. "Shared Event Composition/Decomposition in Event-B". In: *9th International Symposium on Formal Methods for Components and Objects, FMCO*. Vol. 6957. LNCS. 2012, pp. 122–141.

[110] Renato Silva, Carine Pascal, Thai Son Hoang, and Michael Butler. "Decomposition tool for Event-B". In: *Software: Practice and Experience* 41.2 (2011), pp. 199–208.

[111] Neeraj Kumar Singh. "EB2ALL: an automatic code generation tool". In: *Using Event-B for Critical Device Software Systems*. Springer, 2013, pp. 105–141.

[112] Neeraj Kumar Singh. *Using Event-B for Critical Device Software Systems*. Springer, 2013.

[113] Graeme Smith. *The Object-Z Specification Language*. Springer, 2012.

[114] Colin Snook and Michael Butler. "UML-B: Formal modeling and design aided by UML". In: *ACM Trans. on Software Engineering and Methodology* 15.1 (2006), pp. 92–122.

[115] Colin Snook and Michael Butler. "UML-B and Event-B: an integration of languages and tools". In: *IASTED International Conference on Software Engineering, SE*. 2008, pp. 336–341.

[116] Alfred Tarski. "The semantic conception of truth: and the foundations of semantics". In: *Philosophy and Phenomenological Research* 4.3 (1944), pp. 341–376.

[117] Alfred Tarski. "On Some Fundamental Concepts of Metamathematics." In: *Logic, Semantics and Metamathematics. Edited and translated by JH Woodger* (1956).

[118] Simon Thompson. *Haskell: the craft of functional programming*. Vol. 3. Addison-Wesley, 1999.

[119] Qi Wang and Tim Wahls. "Translating Event-B machines to database applications". In: *12th International Conference on Software Engineering and Formal Methods, SEFM*. Vol. 8702. LNCS. 2014, pp. 265–270.

[120] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: getting your models ready for MDA*. Addison-Wesley, 2003.

[121] John B. Wordsworth. *Software Development with Z: a Practical Approach to Formal Methods in Software Engineering*. Addison-Wesley, 1992.

[122] Steve Wright. "Automatic generation of C from Event-B". In: *Workshop on integration of model-based formal methods and tools*. 2009, p. 14.