USING LOGIC-BASED APPROACHES TO EXPLORE SYSTEM ARCHITECTURES FOR SYSTEMS ENGINEERING

A Dissertation
Presented to
The Academic Faculty

by

Aleksandr A. Kerzhner

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
George W. Woodruff School of Mechanical Engineering

Georgia Institute of Technology
August 2012

USING LOGIC-BASED APPROACHES TO EXPLORE SYSTEM ARCHITECTURES FOR SYSTEMS ENGINEERING

Approved by:

Dr. Christiaan J. J. Paredis, Advisor
School of Mechanical Engineering
Georgia Institute of Technology

Dr. Leon F. McGinnis
School of Industrial and Systems Engineering
Georgia Institute of Technology

Dr. Berdinus A. Bras
School of Mechanical Engineering
Georgia Institute of Technology

Dr. Godfried L. Augenbroe
College of Architecture
Georgia Institute of Technology

Dr. Charles Eastman
College of Architecture
Georgia Institute of Technology

Date Approved: April 23, 2012

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support, guidance, and friendship of an entire community of people. I cannot describe it in any other way, from many who passed through Georgia Tech the same time I was here, graduating and leaving for bigger and better things, and those who have been here the entirety of my journey.

First I would like to thank my advisor Dr. Chris Paredis for his continued insight and support, for always keeping an open mind but also staying critical, and for always showing me there's no substitute for hard work. I would also like to thank Dr. Leon McGinnis for serving on both this committee and my master's committee, for always reminding me to keep an eye on practical matters, and reminding me it is important to ask tough questions. I would also like to thank the rest of my committee, Drs. Bert Bras, Godfried Augenbroe, and Charles Eastman for their guidance and advice.

I would also like to thank Richard Malak and Stephanie Thompson for being excellent role models when I was just beginning my graduate career, along with the rest of the members of MARC 266 and MARC 264: Jonathon Jobe, Tommy Johnson, and Aditya Shah for their support and insight and also for prior work that provided the foundation for this dissertation; Benjamin Lee for always being a good friend, even when I was not; Jackie Branscomb. Kevin Davies, and Brain Taylor for their friendship, support; Bergen Helms, Sebastian Herzig, Axel Reichwein, and Ahsan Qamar for giving the lab some international flair; and Isabelle Bouchard, Jayme Walton, Kevin Wright, and Marc Pare` for providing the enthusiasm and dedication that can only spring from undergraduate researchers.

In addition, I would like to acknowledge my many (former) labmates in the (now disbanded) Systems Realization Laboratory: Jamal Wilson, John Reap, Greg Graf, Nathan Young, Amit Jariwala, Jane Kang, Jiten Patel, Dazhong Wu, Yuri Romaniw, Andrew Hyder, Patrick Chang, Julie Bankston, and many more that I am sure I have forgotten. I also owe my gratitude to the rest of my labmates from the Model-Based Systems Engineering Center: Edward Huang, Kysang Kwon, George Thiers, and Ola Batarseh.

I would also like to thank my compatriots in the Georgia Tech-branch of the Center for Compact and Efficient Fluid Power (CCEPF) and also the members of the Intelligent Machine Dynamics Lab (IMDL) for their friendship and support: Hannes Daepp, Mark Elton, Nick Earnhart, Heather Humphreys, Brian Post, Aaron Enes, and Ryder Winck.

Maybe most importantly, I would like to gratefully acknowledge the ERC for Compact and Efficient Fluid Power, supported by the National Science Foundation under Grant No. EEC-0540834 for funding this work.

Finally, I would like to thank my mother, Inna Kerzhner-Haller, for her love and support; without her none of this would be possible.

TABLE OF CONTENTS

Page

Acknowledgements iii

List of Tables xii

List of Figures xiii

Summary xviii

Chapter 1: Introduction 1

1.1 What is this Research About? 1

1.2 Why is this important? 4

1.3 Why is this challenging? 6

1.4 Desired Characteristics of the Approach 10

1.4.1 Guide the designer in making rational decisions 10

1.4.2 Based on Computer-Interpretable Models 11

1.4.3 Efficient Search 12

1.4.4 Flexible formulation 14

1.4.5 Knowledge reuse at different levels of abstraction 15

1.4.6 On adding value 16

1.5 Gap & Vision 18

1.6 Research Questions 21

1.6.1 Information Models and Domain Specific Language 22

1.6.2 Model Libraries & Model Transformations 24

1.6.3 Mixed-Integer Linear Programming 25

1.6.4 Managing Complexity and Supporting Scalability 28

1.7 Expected Contributions 30
1.8 Investigation Roadmap 31
Chapter 2: Prior and Related Work 35
2.1 Current Systems Engineering Practice 35
2.1.1 Systems Engineering Processes 35
2.1.2 System Architecting 42
2.1.3 Software Architecting 44
2.2 Evaluation and Decision Making 44
2.2.1 Qualitative approaches 45
2.2.2 Sizing techniques 46
2.2.3 A Foundation for Modeling Architecture Explorations 47
2.2.4 Surrogate Models 48
2.2.5 Predictive Models 49
2.2.6 Optimization in Systems Design 49
2.3 Computational Design Synthesis 50
2.3.1 Function-Based Approaches 51
2.3.2 Grammatical Approaches 52
2.3.3 Constraint-Based Approaches 54
2.3.4 Adaptation-Based Approaches 55
2.3.5 Knowledge Capture 56
2.3.6 Searching the Design Space 57
2.4 Summary 58
Chapter 3: Representing Architecture Exploration Problems 60

3.1 Prior and Related Work in Modeling Designer Knowledge Explicitly 64
3.1.1 Capturing Variants 64
3.1.2 Domain Specific Languages 66
3.2 Foundation for Modeling Architecture Exploration Problems 68
3.3 What is an Architecture Selection Decision? 72
3.4 Defining a Language for Architecture Selection Decisions 76
3.4.1 Defining Tests 85
3.4.2 Defining the Space of Solutions 89
3.4.3 Capturing Domain Knowledge 94
3.5 Discussion 97
3.6 Summary 99
Chapter 4: Model Libraries and Composition 100
4.1 Prior Work in Modularity and Composition 102
4.2 Capturing reusable Analysis Knowledge in a Model Library 105
4.2.1 A Library of Components 107
4.2.2 A Library of Analyses 109
4.2.3 A Library of Aspects 109
4.2.4 Fine-Grained Design-Analysis Relationships 110
4.3 Implementation in SysML 112
4.3.1 Component taxonomy 114
4.3.2 Aspect taxonomy 118
4.3.3 Library of Analysis Models 119
4.3.4 Establishing Fine-Grain Mappings 121

4.3.5 Parameter Maps 121
4.3.6 Interface Maps 123
4.4 Automated Composition of Analysis Models 123
4.4.1 An Illustrative Example 124
4.4.2 Creating the Analysis Model 127
4.5 Referencing external models in a model library 133
4.6 Discussion 135
4.7 Summary 137
Chapter 5: Architecture SElection Using Mathematical Programming 138
5.1 Desired Characteristics of the Search Process 140
5.2 Choice of Solution Approach 143
5.2.1 Constraint Satisfaction Approaches 148
5.2.2 Boolean Satisfaction 150
5.2.3 Mathematical Programming 152
5.2.4 Agent-Based Approaches 155
5.3 Structure of an Architecture Exploration Problem in MIP 155
5.4 Defining the Structure 159
5.4.1 Describing System Behavior 160
5.4.2 Optional Constraints 164
5.4.3 Interpolation 165
5.5 Representing Algebraic Analysis Models in SysML 171
5.6 Discussion 174
5.7 Summary 176

Chapter 6: Problem Transformation 177

6.1 Defining Model Transformations 179

6.2 Transformation Issues 182

6.2.1 Practical Considerations 184

6.3 Transformation Process – First Stage 187

6.4 Generating AIMMS Code 199

6.5 Import-Export SysML 200

6.6 Summary 201

Chapter 7: Excavator Example 203

7.1 Defining the example in SysML 207

7.1.1 Comparison to other architecture exploration problems 215

7.2 Outline of Optimizations 217

7.3 The Mathematical Programming Framework 219

7.4 Verification Examples 223

7.5 Full Excavator Example 232

7.5.1 Optimizing for Total Cost 235

7.6 Constrained Example 237

7.7 Unsized Solutions 242

7.8 Comparison with Similar Approaches 243

7.9 Summary 251

Chapter 8: Contributions, Limitations and Open Questions 252

8.1 Review of the Research 252

8.2 Summary of Contributions 259
8.2.1 Modeling Architecture Exploration Problems 259
8.2.2 Architecture Exploration and Computational Design Synthesis 260
8.2.3 Mathematical Programming 261
8.3 Limitations 262
8.3.1 Cost of Modeling 262
8.3.2 Creating Model Libraries 263
8.3.3 Uncertainty 263
8.3.4 Scalability 264
8.3.5 Accuracy of Analyses – Using Only Linear Constraints 264
8.3.6 Non-Unique Architectures 266
8.3.7 Debugging the Formulation 266
8.4 Practical Implementation 267
8.5 Open Questions and Opportunities for Future Research 268
8.5.1 Practical Aspects of Modeling an Architecture Selection Decision 268
8.5.2 Practical Aspects of Solving an Architecture Selection Decision 270
8.5.3 Extensions 271
8.5.4 Informing Designers 272
8.6 Summary 272
Appendix A : Model Libraries 275
A.1 Component-level structural models. 275
A.2 Component-level analysis models. 277
A.3 Functional Units 287

A.4 Connection Templates 289

Appendix B : Sample Aimms Code 295

References 338

LIST OF TABLES

Page

Table 7.1: Results from optimization runs where number of points in the interpallent are varied..... 232

Table A.1: Commercial off-the-shelf pumps. 276

Table A.2: Commercial off-the-shelf cylinders. 276

Table A.3: Commercial off-the-shelf engines. 276

Table A.4: Maximum torque (Nm) for a given normalized speed for the engines..... 276

Table A.5: Fuel consumption (kg/W) for a given normalized speed for the engines.... 277

LIST OF FIGURES

Page

Figure 1.1: Conceptual overview of a multi-staged solution approach. 29

Figure 3.1: A combination of UML profiles and metamodel based technologies..... 68

Figure 3.2: Breakdown of a Decision into its basic features. 70

Figure 3.3: Decision Process adapted from Hazelrigg. (Hazelrigg, 2012) 72

Figure 3.4: Classic description of an architecture selection decision. 74

Figure 3.5: Modular representation of the architectures considered in the architecture selection decision. 75

Figure 3.6: SysML Profile that defines the additional stereotypes needed to represent an architecture selection decision. 78

Figure 3.7: Profile for additional constructs added for defining the architecture selection decision. 80

Figure 3.8: SysML profile for defining testable requirements. 83

Figure 3.9: Relationship between testable requirements and test cases..... 84

Figure 3.10: Simple test case. The system under test is connected to two testing probes (sensors). 86

Figure 3.11: Utilizing readSelf and readStructuralFeature actions to compare the value of var1 to a test value. 88

Figure 3.12: Simplified definition of a potential system 91

Figure 3.13: Simplified definition of a potential subsystem..... 92

Figure 3.14: Simplified connection template between two components 93

Figure 3.15: An amalgamation of models related to the structural cylinder. These different relationships are represented using *AssociationBlocks*. 95

Figure 4.1: Profile for capturing correspondences between structure and analysis models. 114

Figure 4.2: A partial view of the Component hierarchy. 115

Figure 4.3: Component hierarchy of pumps 116

Figure 4.4: Sub-system taxonomy, sometimes referred to as functional units. 117

Figure 4.5: Relation between the subsystem taxonomy and the component taxonomy . 118

Figure 4.6: Package structure for aspect taxonomy. 119

Figure 4.7: Overview of algebraic behavior analysis models. 120

Figure 4.8: AssociationClass linking structural model of a double acting cylinder to an analysis model. 121

Figure 4.9: Parameter map between parameters of the double acting cylinder and the structural model. 122

Figure 4.10: Interface map between the ports of the cylinder's analysis model and the structural model. 123

Figure 4.11: Example hydraulic circuit's structural model. 126

Figure 4.12: Test context describing the simulation and analysis model to be generated. 127

Figure 4.13: Resulting analysis model. 133

Figure 4.14: Profile for defining external models and libraries. 134

Figure 4.15: Resulting Modelica code. 135

Figure 5.1: Visualization of the design space. 141

Figure 5.2: One dimensional interpolation. 167

Figure 5.3: Unscaled interpellant. The red points are true data and the surface is the interpellant. 169

Figure 5.4: Unscaled versus scaled interpellants. 171

Figure 5.5: Profile for representing algebraic models in SysML..... 174

Figure 6.1: Generic structure of model transformations. Adapted from Czarnecki (Czarnecki, 2006)..... 180

Figure 6.2: Relations between the QVT languages (Object Management Group, 2007). 181

Figure 6.3: The structural definition of a simplified model fragment 188

Figure 6.4: The internal definition of the simplified model fragment 189

Figure 6.5: The internal definition of the *OCPowerUnit* functional unit. 191

Figure 6.6: Components resulting from flattening of the original space definition..... 193

Figure 6.7: Flattened view with corresponding connectors. 195

Figure 7.1: Excavator, taken from (Haga, 2001) 203

Figure 7.2: A Requirement Diagram for the Hydraulic System that also includes the proposed testable requirements..... 209

Figure 7.3: Test context for excavator subsystem including the IBD for the hydraulic subsystem. 210

Figure 7.4: Test cycle for the actuation subsystem..... 211

Figure 7.5: Visualization of the test cycle for each cylinder. 212

Figure 7.6: Block definition diagram representing the entire excavator structure. 213

Figure 7.7: SysML Internal Block Diagram of the excavator showing the partially specified hydraulics subsystem. 214

Figure 7.8: Structure for the verification example. Only one type of each component is included. 223

Figure 7.9: Resulting architecture from simple verification example. 225

Figure 7.10: Verification example with two of each component. 226

Figure 7.11: Internal structure of the second verification example. 227

Figure 7.12: Verification example with one pump and one prime mover. 229

Figure 7.13: Verification example with two pumps and two prime movers. 230

Figure 7.14: Architecture resulting from optimization of cost. 235

Figure 7.15: Architecture found when minimizing life-time cost. 237

Figure 7.16: Structure of the constrained example. 239

Figure 7.17: Internal structure of the constrained example. 239

Figure 7.18: Architecture found when minimizing cost. 240

Figure 7.19: Architecture found when minimizing fuel consumption. 242

Figure A.2: Engine analysis model 279

Figure A.3: Illustration of the valve's schematic.. 280

Figure A.4: Open center valve with neutral pass through analysis model. 282

Figure A.5: Schematic for the closed center valve. 283

Figure A.6: Analysis model for the closed center valve. 284

Figure A.7: Fixed-displacement pump analysis model. 286

Figure A.8: Fixed-displacement power unit. 287

Figure A.9: Open center valve block. 288

Figure A.10: Closed center valve block. 289
Figure A.11: Connection templates represented in the component library 290
Figure A.12: Connection template between a fixed-displacement power unit and an open-center valve block. 291
Figure A.13: Connection template between an open-centered valve block and another open-center valve block. 292
Figure A.14: Connection between a fixed-displacement power unit and an open-centered valve block 292
Figure A.15: Connection between a prime mover (engine) and a (generic) power unit. 292
Figure A.16: Connection between a cylinder and a (generic) valve block. 293
Figure A.17: Connection between a motor and a (generic) valve block. 294

SUMMARY

This research is focused on helping engineers design better systems by supporting their decision making. When engineers design a system, they have an almost unlimited number of possible system alternatives to consider. Modern systems are difficult to design because of a need to satisfy many different stakeholder concerns from a number of domains which requires a large amount of expert knowledge. Current systems engineering practices try to simplify the design process by providing practical approaches to managing the large amount of knowledge and information needed during the process. Although these methods make designing a system more practical, they do not support a structured decision making process, especially at early stages when designers are selecting the appropriate system architecture, and instead rely on designers using ad hoc frameworks that are often self-contradictory.

In this thesis, a framework for performing architecture exploration at early stages of the design process is presented. The goal is to support more rational and self-consistent decision making by allowing designers to explicitly represent their architecture exploration problem and then use computational tools to perform this exploration. To represent the architecture exploration problem, a modeling language is presented which explicitly models the problem as an architecture selection decision. This language is based on the principles of decision-based design and decision theory, where decisions are made by picking the alternative that results in the most preferred expected outcome. The language is designed to capture potential alternatives in a compact form, analysis knowledge used to predict the quality of a particular alternative, and evaluation criteria to differentiate and rank outcomes. This language is based on the Object Management

Group's System Modeling Language (SysML). Where possible, existing SysML constructs are used; when additional constructs are needed, SysML's profile mechanism is used to extend the language.

Simply modeling the selection decision explicitly is not sufficient, computational tools are also needed to explore the space of possible solutions and inform designers about the selection of the appropriate alternative. In this investigation, computational tools from the mathematical programming domain are considered for this purpose. A framework for modeling an architecture selection decision in mixed-integer linear programming (MIP) is presented. MIP solvers can then solve the MIP problem to identify promising candidate architectures at early stages of the design process. Mathematical programming is a common optimization domain, but it is rarely used in this context because of the difficulty of manually formulating an architecture selection or exploration problem as a mathematical programming optimization problem. The formulation is presented in a modular fashion; this enables the definition of a model transformation that can be applied to transform the more compact SysML representation into the mathematical programming problem, which is also presented. A modular superstructure representation is used to model the design space; in a superstructure a union of all potential architectures is represented as a set of discrete and continuous variables. Algebraic constraints are added to describe both acceptable variable combinations and system behavior to allow the solver to eliminate clearly poor alternatives and identify promising alternatives.

The overall framework is demonstrated on the selection of an actuation subsystem for a hydraulic excavator. This example is chosen because of the variety of potential

architecture embodiments and also a plethora of well-known configurations which can be used to verify the results.

CHAPTER 1:

INTRODUCTION

1.1 What is this Research About?

This research is focused on helping engineers design better systems by supporting their decision making for choosing an appropriate system architecture. When engineers design a system, they have an almost unlimited number of possible system alternatives to consider. Designers undertake a systematic design process to prune this space of system alternatives to arrive at a single system specification. This specification consists of two parts: the specification of the architecture and the specification of the components in this architecture. The architecture describes the types of components or subsystems that are contained in the system, their interfaces, and how the components are connected through these interfaces. The specification of components provides more detailed information about each component's sizing.

During this process, designers need to make two different *types* of decisions:

- Architecture Selection Decisions: Decisions between different *types* of systems. In this work, this is described as selecting between different system architectures. These can be considered as decisions made between a discrete set of potential architecture alternatives.
- Requirements Allocation Decisions: Often referred to as requirements flow-down or component sizing, these decisions determine the appropriate specifications or sizes of the components of a particular system architecture (Often referred to as sizing the system). Here, the decision is made over a continuous space of choices, although sometimes component availability can make certain discrete choices

clearly desirable. Some previous work refers to requirements allocation decisions as compromise decisions (e.g., (Bras, 1993, Karandikar, 1989, Shupe, 1987)) or parametric design. Labeling these as requirements allocation decisions is based on previous work by Malak and Paredis (Malak, 2010) .

Usually these decisions are made in a top-down, hierarchical progression. First, designers make decisions about the type of architecture using their expertise and design intuition. After the architecture has been selected, they drill down into the requirements allocation and select the appropriate components and sizes.

As an example, when designing a car's power train, a designer can choose a purely mechanical implementation, a purely electric implementation, or a hybrid implementation. Simply choosing the purely mechanical architecture is not sufficient to specify the system. The designer also needs to choose sizing parameters such as the volume, power, or torque of the engine or the gear ratios of the transmission. These choices are usually made after the architecture has been selected. The choice of power train is between discrete choices; it would be meaningless to choose a system which lies "between" these alternatives. On the other hand, the parameters such the gear ratio or engine torque can be varied continuously, although the parameters cannot be selected independently and are constrained by the available technology.

The process that designers take to arrive at the system specification is *architecture exploration*. As part of this exploration, architectures are synthesized and evaluated to move toward the "best" system alternative. Going back to the previous example, it would be difficult to choose between the mechanical or electrical architecture without considering potential embodiments; while looking at the structure of the architecture may

allow designers to make generic statements about its performance, these support only obvious differentiations. In order to evaluate an architecture more accurately, the designer needs to determine and evaluate the "best" instance of that architecture. Since changes in the design context can affect the best instance, to quantitatively differentiate architectures rationally a nested optimization is needed at each step of the architecture exploration process to choose the best instance of each architecture.

Although this is a difficult process, practicing designers are very capable of designing good systems; they routinely design cars, airplanes, construction machinery, computers, robots and other complex devices. The problem is that shifting consumer preferences have placed additional expectations on the performance of these systems making them more difficult to design; therefore, designers are finding it more difficult to complete a design project without cut corners or budget overruns (Sage, 2000a). As a means to reduce the overall time of the design process, designers try to quickly make system selection decisions and then expend most of their effort on requirements allocation decisions. It is not that designers lack the necessary knowledge to make the system selection decisions; instead they lack the necessary tools to apply this knowledge in an efficient manner during their design processes.

The goal of this research is to provide designers with computational tools, both for modeling and performing architecture explorations, so they can get more out of their limited resources. This will allow designers to more broadly explore the architecture space early on in the design process and also help them design better systems by allowing for more rational and quantitative decision-making earlier in a project than is currently practical.

1.2 Why is this important?

The underlying assumption in this research is that architecture exploration is important because more exploration leads to better designs. There are three possibilities for how performing additional exploration will affect the outcome of the design process: the quality of the final design will decrease, the quality of the final design will stay the same, or the quality of the final design will increase. For the sake of this discussion, assume that the quality of a design can be quantified. There is always the opportunity that more exploration will decrease the quality of the final design; since there is uncertainty involved when choosing the final design, by considering more designs there is the possibility that a new design will appear better using the defined metrics but perform worse in reality. A bad outcome can be the result of any decision, what is important is the quality of the decision. Another possibility is the additional exploration does not change the choice of final design because the final design is in the original set of designs. Although the final design does not change, this should increase designer confidence in the final design. The other possibility is that increased exploration will identify a new design that is better than the original design. The probability of the case where the final design is worse can be reduced by improving the quality of the analyses used during the design process, although this is not considered as part of this investigation. This leaves the other two cases where either that the exploration will not change the final design or that the exploration will result in a new design that is better than the original design. Both of these situations would add value to the design process, but would also incur the additional exploration cost. This suggests that a tradeoff exists between the space of solutions explored and the time and cost of exploration those solutions, and also that

performing an architecture exploration adds value as long as it can offset the time and cost of preforming the exploration.

Performing such a broad exploration is important in many product domains. It is important in domains where there are a large number of plausible architectural alternatives, none of which is clearly better in all contexts (dominant). This, for instance, can be the result of considering existing technologies in new applications. One example is the recent adoption of electrical components in hydraulic systems, which traditionally have included mainly mechanical and hydraulic components. Whereas in the past, designers could rely on prior experience in these domains, the influx of new technology requires designers to gain additional knowledge by evaluating new options and comparing them with previous alternatives. Until the holes in the domain knowledge have been filled, a thorough exploration of the new, previously unexplored architectures is important. Given the rate of technology adoption, these "holes" in the domain knowledge may never be filled and as some are filled, others may appear.

In addition, competitors can often differentiate themselves by creating innovative products that go beyond simply resizing previously used architectures. Architecture exploration in these domains needs to be efficient because they usually require short development cycles as a result of changing technology and a need to reach the market first.

Although overall, quantitative data to support the value and impact of improving the quality of architecture selection decisions is limited, one source of evidence is in the success of Toyota Motor Company; this success is attributed in some part to their unique approach for making architecture selection decisions. The decision making process at

Toyota is usually referred to as set-based concurrent engineering; in this approach multiple potential solutions (often the choice of appropriate architecture) are investigated during the design process and the selection between them is delayed (Sobek Ii, 1999). The suggestion is that by considering more solutions than is common in other firms, Toyota is able to achieve better designs; this suggests that picking the appropriate architecture is important, not just optimization of the component sizings.

Although broader exploration of the design space is likely to lead to better resulting designs, the difficulty is more exploration results in the design process incurring additional cost. Therefore, even with several compelling reasons for broad exploration, in current design practice designers only investigate a few candidate architectures. They select these architectures based on previous experience (Gero, 1996) or by quickly narrowing the solution space using focused ideation methods (Pahl, 2007). Most of the focus and effort is then applied to choosing the appropriate sizing for the components within these architectures. With current practices, broad architecture exploration is not pursued because it is too time-consuming or prohibitively expensive. Instead, ideating possible architectures is left almost entirely to a human designer with minimal computational assistance. By providing designers with better computational tools, more of the design space can be explored before a decision is made as to the correct architecture.

1.3 Why is this challenging?

The current state of the art includes a number of well accepted systems engineering processes available to support the design of a system. These processes provide clear steps that designers can go through when designing a system. Although

each process is slightly different (Buede, 2000, Parnell, 2011), the steps can be generalized as:

1. Identification of the performance objectives and requirements: During this step, the various stakeholders involved with the system come to a consensus of how the system should perform, what are the desired functions and behavior, and so on.
2. Preliminary Design: Designers focus on making high-level system selection decisions, such as the selection of the architecture. This phase is often broken down into multiple steps including the definition of a logical (sometimes referred to as a platform independent or functional) architecture and then the synthesis of the actual physical architecture based on this logical architecture.
3. Detailed Design: Here, the focus is on the design of individual system components, which also includes writing the necessary software to control the system.
4. Integration of components into the system: After the appropriate components are selected, purchased, or manufactured, they need to be integrated together into a completed system.
5. Testing of the final system: The system is tested thoroughly to insure it is capable of meeting the requirements prescribed in the first step. This process can be very time-consuming and the systems deficiencies that are identified during this step are costly and difficult to fix.

This investigation is primarily focused on the 2nd step where designers are focused on making system selection decisions. Providing designers support early on in the design process is challenging because of the scope of modern design problems and the inherent

uncertainty present during early design stages. The fundamental problem is how to model the knowledge associated with an architecture exploration then efficiently utilize that knowledge to support designers' decision making.

During traditional design processes, most of the problems that arise are because of organizational complexity, not direct technology concerns affecting individual subsystems or specific physical science areas. When trying to provide computational support in this area, the issues are driven by the same problem of managing complexity and insuring methods can scale sufficiently as to be useful. Many of the challenges that arise are driven by the sheer scale of the design problem. Designers have a myriad of potential architectures to consider, and for each of the architectures there is an almost infinite combination of applicable component sizings.

Because of the size of this space, many traditional design exploration processes only consider it implicitly. Instead, the desired architecture is chosen in an ad hoc process by gathering a large group of highly-skilled domain experts who use their knowledge and experience to ideate and evaluate a handful of potential candidate solutions.

Because of the diverse performance requirements placed on a system, many aspects of a system need to be evaluated; therefore, in addition to the need for domain knowledge that spans a large number of different architectures, there is a need for in-depth knowledge about each architecture.

As a result, there is a large amount of domain specific knowledge that these designers are using during the process, but this knowledge is often only available in the designers' minds. This domain knowledge is often varied in form and difficult to represent; the challenge is that without capturing this knowledge in a form that is

computer interpretable it is very difficult for computational tools to support the design process.

In addition to the large amount of domain knowledge that comes with such a large space of potential alternatives, the size of the space makes it very difficult to search it effectively. In addition, as alluded to in Sections 1.2, the search process is complicated by the difficulty of distinguishing between different candidate architectures. Simply generating a large number of solutions is difficult for human designers to process and it is also not clear how well these generated solutions truly span the space of all potential alternatives.

Analyzing these alternatives is also difficult; using current approaches designers expend a large amount of effort on creating system-level analysis models that can be used to size a particular architecture. Doing this during the preliminary design stage can greatly slow the process because different analyses are needed for each system architecture.

As can be seen from the description of the systems engineering process, after the preliminary design stage where many of the system-level decisions are made, there is still a significant design effort remaining. Also, most of the effort and resources that are allocated to the design process will be used after this stage. As a result, at the preliminary design stage there is significant uncertainty about the performance of any potential architecture. Whichever architectures are selected in this phase (in almost all traditional processes, only one architecture is selected) will receive significant attention and design effort to insure that the final system is able to meet the prescribed requirements. These challenges make providing computational tools difficult.

1.4 Desired Characteristics of the Approach

Considering these challenges, this section provides an overview of the desired characteristics for an approach to efficiently and effectively utilize domain knowledge in selecting between architectures during preliminary design. These characteristics are a matter of perspective and different people may identify different characteristics or assign the emphasis differently.

1.4.1 Guide the designer in making rational decisions

The most important characteristic of any approach is providing support for the designers to make better system selection decisions. Therefore, the approach should be internally consistent and not result in selections that are contrary to stated preferences or the available knowledge. Decision theory provides well-established principles for decision making (Hazelrigg, 2012). The decision process can be broken into four main steps (which are illustrated in Figure 3.3 on p. 72):

1. Formulate the decision in terms of a solution-independent objective. There are a number of approaches for creating this objective, such as Multi-Attribute Utility Theory (Keeney, 1976) or Value-Driven Engineering (Castagne, 2009).
2. Identify the different alternatives that are being considered.
3. Predict the potential consequences or outcomes of choosing an alternative.
4. Evaluate each alternative outcome relative to the objective and identify the most-preferred, thereby selecting the most preferred alternative.

These steps can help inform the correct structuring of an architecture selection decision where the designer is making a decision between multiple potential architectures, as in an architecture exploration. Guaranteeing an approach truly helps

designers make consistent decisions is very difficult. Because of the scale of an architecture exploration problem, capturing and applying all available knowledge to predict the potential outcomes is impossible with current technology. For any approach to be practical, significant simplifying assumptions are needed. Making simplifying assumptions results in models with incomplete information which can lead to unsound decision-making. Therefore, it is important to also understand the potential impact these simplifying assumptions may have on the final result. Also, it is likely that the architecture selection decision will need to be broken down into a set of simpler sequential decisions.

Another significant issue is that decision theory is only applicable for situations where there is a single decision maker and there is no scheme for aggregating together the preferences of multiple decision-makers and still guaranteeing rationality and self-consistency (Arrow, 1963). As will be discussed more thoroughly in Chapter 3, by explicitly expressing domain knowledge in a form where it can be reviewed and consensus can be reached by designers along with a definition of the exploration and relevant objective, it may be possible to create a suitable facsimile of a single decision maker. In current processes, these decisions are usually made by teams of engineers using ad hoc selection criteria (Hazelrigg, 2012, Parnell, 2011) and one goal of this work is to drive these processes toward more rational decisions.

1.4.2 Based on Computer-Interpretable Models

Traditional systems engineering processes rely heavily on designer insight or intuition (hence-forth referred to as mental models) during architecture exploration. These mental models often lack the fidelity to truly distinguish between alternatives and

are subject to personal biases. For the kind of complex problem that is common in systems engineering, it is also very difficult for a person to internalize and rationally consider the plethora of choices.

In addition to a reliance on mental models, the architecture exploration is recorded in paper documents. This means that the different stakeholder requirements and objectives and the results of a multitude of analyses are in a form that is difficult to review and apply.

Instead, it is desirable for the approach to capture as much of the architecture exploration as possible in a form where it can be used to support the decision making process. In order to accomplish this, computer-interpretable models are needed. Many of these models take the form of information models which can be used to explicitly define the scope of the exploration, the needed objectives, and analyses.

1.4.3 Efficient Search

Because of the large number of potential solutions, there is a very large search space within which the best system architecture needs to be identified. Part of the difficulty is in evaluating which architecture is truly the best. The other is in searching this very large space in an efficient manner where results can be delivered to designers in a reasonable amount of time. As mentioned earlier, a design project is constrained by limited resources, which include available computer time. Also, any solver chosen needs to be able to handle this large search space.

Although in traditional processes designers are seen as ideating the potential architecture, this can be thought of as designers applying their domain specific

knowledge to first eliminate architectures that they recognize as clearly poor and then selecting the best remaining alternative.

In order to compare between different architectures, the performance of each architecture needs to be predicted. Here, varying amounts of designer knowledge can be used in predicting the expected performance of the architectures and trying to compare them. In early design stages, the performance is usually described qualitatively by making generic statements of how each of the architectures will perform (Parnell, 2011, Sage, 2000a). As the process progresses, these performance estimates shift toward quantitative predictions based on more concrete analyses. By slowly increasing the fidelity of the analyses that designers perform, they are attempting to more efficiently utilizing their scarce resources by managing the scope of the problem.

Taking this into consideration, a similar search approach could be desirable where early on in the solution process only a subset of the domain knowledge would be used to create analysis which would be useful in eliminating clearly inferior designs (for example, one could eliminate architectures that would not provide desired functionality given the desirable objectives and context).

Then, more effort could be used in evaluating more promising architectures. This could entail using a variable-fidelity or variable-accuracy approach which is quite common for problems where analyses take a long time to execute (Thompson, 2010). The difference here is that instead of simply using multiple existing models that produce results with different accuracies, these models will need to be constructed for each potential architecture.

1.4.4 Flexible formulation

Complex systems appear in a large number of domains, from aerospace to construction equipment to computers. There is a very diverse set of potential components and connections that can appear in these systems.

Any approach must be flexible enough to handle these very diverse constructs and also facilitate the addition of new components and connections as technology advances. This eliminates approaches where most of the domain knowledge is hard-coded and difficult to change because applying such approaches is unlikely to be practical.

The formulation also needs to be accessible to domain experts so they can encode their own domain knowledge, because it is unlikely that the captured knowledge will sufficiently cover their domain and asking non domain experts to capture this knowledge increases the opportunity for errors and omissions.

To provide support for these process, it is important to be able to represent the definition of the architecture exploration problem in a generic and flexible manner where a large amount of disperse domain knowledge can be incorporated.

Also, this representation needs to be sufficiently solution independent so that the same domain knowledge can be reused at different levels of abstraction during disperse phases of the solution process. Knowledge about the problem also needs to be captured in a form that is independent from a particular architecture instance so that different aspects of the problem definition can be composed to evaluate a particular architecture alternative.

1.4.5 Knowledge reuse at different levels of abstraction

During an architecture exploration, there is a need for a large number of analysis models. Usually, for each individual architecture a different analysis is needed. Even if this is not the case and multiple architectures can be considered with the same analysis model, a large amount of domain knowledge will need to be synthesized to create these analysis models. Also, as discussed previously, the scope and nature of architecture exploration problems is constantly changing with the addition of new customer expectations or new available technology. As a result, it is important the designers are able to reuse any knowledge they have captured in computer-interpretable models.

It is also desirable to reuse domain knowledge across multiple architectures. Many current systems engineering approaches express domain knowledge relative to a single architecture (Estefan, 2007). Instead, since architectures are composed of common components modularity could be utilized to capture domain knowledge at the component-level and then composed into system-level models. Also, this would allow designers to tweak the exploration problem early on when the design process can be plagued by shifting expectations and objectives.

In order to support composability, the component-level models need to be declarative in nature. There are two main approaches for defining and executing analysis models: a declarative approach and an imperative approach. In an imperative approach (also referred to as a procedural approach), the execution sequence needed to solve the models is explicitly captured. This means that the procedure for executing the model is included in the definition of the model. Often, the definition of the model is implicitly captured in the computer code. For instance, consider MATLAB (Mathews, 1998)

models where the solution process simply involves executing each line of code sequentially. In such cases, it is difficult to simply compose multiple imperative code fragments into working code. Because the execution order is explicitly defined in the code fragment fixing the input/output relationship between the variables in the code, if the code fragment is used in a different context it may no longer be applicable. For instance, if a piece of code computes the pressure produced by a pump when given inputs of torque and angular velocity, this code would not be applicable for computing the torque or angular velocity given the pressure. Potentially, a root-finding algorithm could be included in the code execution to reverse the causality, but this would add unnecessary complexity to the model. On the other hand, in the declarative approach the analysis model is defined without this explicit sequence, instead only the various equations (or constraints) and variables are defined, and the simulation procedure determines the appropriate sequence for solving or simulating these models. This makes is possible to compose multiple model fragments; how to solve these fragments can then be determined by the solver.

1.4.6 On adding value

When considering the previously enumerated characteristics, it is important to consider how these add value to a design process in terms of allowing designers to choose better designs and sustain a competitive advantage while also considering the additional cost of each of these characteristics and insuring that the net result is a positive one.

For instance it is desirable that a design method is self-consistent and rational, but this usually makes the method more difficult to implement in practice (Hazelrigg, 2012).

Eliciting the designer knowledge necessary and handling uncertainty can come at a high cost, especially since designers are not adequately trained in statistics, uncertainty, or decision theory. In current practice, one of the key enablers of the adoption of a design methodology is that it is easy to apply with principles that are easy to understand, even if they are not correct or rigorous (Hazelrigg, 2012).

Capturing the relevant designer knowledge in computer-interpretable models increases the cost of modeling the problem, although it enables knowledge reuse. There can be significant additional cost in training designers to represent their knowledge in models and also significant overhead in creating these models. This is especially true if a flexible formulation is used because it is often the case that designers must use generic constructs to represent their knowledge, which is more cumbersome than if these constructs were tailored to be domain-specific. Also, in order for these models to be reusable across multiple iterations of the same project or even different projects, they must be semantically rich and syntactically consistent enough to insure they are correctly interpreted. Although reuse can greatly reduce the cost of the modeling effort in future projects, there is also significant overhead in initially creating such reusable models.

To enable an efficient search process, it is usually necessary to constrain the nature of the search space. Usually, efficient search is enabled by making assumptions about this search space. Therefore, although certain search algorithms may be extremely effective on a particular type of problem, formulating that type of problem may require the exclusion of certain domain knowledge.

When each of these characteristics is considered for the approach, it is important that their implementation actually adds value to the design process and that the additional costs incurred are offset by this added value.

1.5 Gap & Vision

This research examines an approach to explicitly model architecture exploration problems using information models and then an approach to transform this representation into a number of analyses that can support designers when performing an exploration. This approach relies on model transformation and composition of information models, and the use of mathematical programming optimization tools to provide the efficient search capability needed to support the exploration process.

Although information modeling is becoming more common in systems engineering with such efforts as Model-Based Systems Engineering (MBSE) (Friedenthal, 2008), in this research the push is toward using these models to support better decision-making. In order to accomplish this, the approach needs to provide designers with the tools to explore more potential architecture alternatives than is currently possible.

There exists a number of computational synthesis approaches focused on the generation of potential solutions, but this goal cannot be achieved through simply exploring more solutions, because if these solutions are poor solutions it is unlikely that the final design will improve in quality. Instead, the assumption is that by exploring and evaluating additional *promising* candidate architectures, there is a greater probability that the designer will choose a better design.

For the approach to focus the investigation on promising solutions, not just possible solutions, it is important to allow designers to encode their knowledge about the domain so that promising solutions can be separated from poor solutions. By providing designers with the means to encode their pre-existing knowledge about the nature of the solution space, they can identify regions that are indeed promising. Depending on the problem context the domain knowledge and promising solution space will change, so it is important that this knowledge is flexible and easy to maintain. To support this, in this research an approach for capturing the domain-specific analysis knowledge (the knowledge needed to analyze a particular system alternative) in information models will be investigated. By using information models, the domain knowledge can be stored in a form that is reviewable and modifiable by the designer which is not true when this domain knowledge is captured with hard-coded custom code as is common in many other approaches.

Previous approaches rely on custom imperative code as a means to encode the domain knowledge needed for the analysis or simulation of different alternatives (Antonsson, 2001, Koza, 2010). In this research, instead of implicitly encoding the representation of the architecture exploration problem in this custom code, the focus is on using declarative models that are composed using model transformations into (potentially multiple) analyses. By separating the problem definition from the analyses that guide the solution process, multiple analyses at different levels of abstraction could be created from the same problem definition.

In order to allow generic transformations, the information models must be defined using a semantically rich language; to support the goals this language must also be

accessible to designers. In order to accomplish this, in this research the definition of a novel domain-specific modeling language will be investigated. This language will be an extension of the Object Management Group's Systems Modeling Language (OMG SysML^{TM} or SysML for shorthand purposes) (Friedenthal, 2008, Object Management Group, 2006). Since SysML is gaining popularity among practicing systems engineers, using it as the basis and extending it means these practitioners will need to learn only a handful of new concepts.

Just defining the problem definition in information models is not sufficient, it is also important to identify the appropriate analyses that should be used to guide the solution process. These analyses can be thought of as applying the designers domain knowledge to guide the exploration. The nature of the solution process also informs the structure of the problem definition because the domain knowledge that is necessary during the process must be encoded. Analysis knowledge is needed to distinguish between different architectures, but capturing this analysis knowledge for every potential architecture is extremely time-consuming. To circumvent this, many approaches only use very generic knowledge to differentiate architectures or focus only on encoding knowledge about the desired structure of an architecture. Although this can provide coarse differentiation of solutions, it is very difficult to include the design context with such approaches and the design context informs the choice of the best architecture.

In this research, an alternative approach is investigated where analysis knowledge is captured at the component-level; to evaluate an architecture, component-level models are composed into system-level analyses. To allow for the composition of models in this

way, the models must be captured in a declarative form that can then be interpreted and modified by a solver.

To perform the exploration at early stages, this research will investigate the use of the mathematical programming optimization tools as a means for efficient search to identify potentially promising solutions. Mathematical programming is chosen because of the availability of high-quality solvers that can efficiently perform very large mathematical programming optimizations and the availability of languages that allow mathematical programming problems to be represented in a form that is solver independent so that multiple solvers (and multiple solution approaches) can be applied to the same problem. Mathematical programming is rarely used in this domain because of the difficulty of manually generating the necessary problem code that can be interpreted by a solver. Instead, many methods create custom solvers that are tied directly to the problem representation. To address this concern, this code will be automatically generated through the use of transformations. Previous work has shown that mathematical programming is relevant to this domain by using small-scale mathematical programming optimizations to support the design of simple chemical networks. In this research, the investigation will focus on applying the technique to much larger system design problems because automatically generating the code will allow the creation of significantly larger optimizations.

1.6 Research Questions

To support architecture exploration processes that can be applied to real-world design problems, the emphasis needs to be on the capture and use of the designer's

domain knowledge. This leads to the following research question that summarizes the motivation of this investigation:

RQ: How should designers best represent, manage, and apply knowledge for efficient exploration of system architectures?

This question is too broad to be answered in a single study. Instead, it can serve as a starting point to identify several more focused research questions. There are four main research questions:

RQ1. How should the designer define an architecture exploration problem?

RQ2. How can domain-specific synthesis and analysis knowledge be captured and organized effectively to allow for composition and reuse?

RQ3. What optimization framework is best suited for identifying promising architectures?

RQ4. How should problem scale be managed?

1.6.1 Information Models and Domain Specific Language

When designers perform an architecture exploration process in typical systems engineering processes, designers do not explicitly define the exploration problem. Instead, the focus is often on explicitly capturing the results of their efforts in design documents so that a large number of stakeholders can understand and internalize the final specification. The main strength of documents, that they are easily accessible to humans

because they require minimal training to use and modify, also makes them difficult for computers to interpret.

One could argue the exploration problem is captured implicitly in the analysis models used during the design process. Although it is true that these analysis models have a particular architecture implicitly encoded, to explore architectures not considered by the particular analysis requires manual modification of the analysis. Also, if a different analysis is needed for a particular architecture, this analysis must also be created manually.

These existing approaches are insufficient for explicitly defining the architecture exploration problem. Instead, the following hypothesis, which corresponds to RQ1, is studied in this research:

H1: Designers can represent their architecture exploration problem in information models as an architecture selection decision consistent with decision theory using a domain-specific language.

Although there is a growing trend of documenting the results of a design process in information models, often referred to as Model-Based Systems Engineering, these models lack the necessary detail to support a designer's decision making process. Instead, a novel representation for architecture exploration problems as selection decisions is presented in Chapter 3. To validate this representation, it is applied to the design of the hydraulic subsystem for a hydraulic excavator in Chapter 7.

This representation bases the definition of an architecture selection decision on the structure of decisions from decision theory. It includes a description of the space of potential solutions, domain knowledge that can be used to predict the performance of

these solutions (so-called analysis knowledge), and also evaluation criteria to allow different solutions to be ranked. These parts of the problem definition are captured in computer-interpretable information models so that model transformations can be applied to transform the appropriate aspects of the problem definition into a particular analysis.

To define the metamodel for the representation, a novel domain-specific language will be defined that extends SysML (introduced in Section 1.5) will be used. SysML is a very generic systems modeling language designed to represent many of the important aspects that go into the definition of an architecture exploration problem. SysML contains concepts for capturing system requirements, behavior, and structure. A major gap in the constructs and best practices that exist with SysML is that previous emphasis has been on documenting systems engineering processes instead of capturing knowledge and utilizing that knowledge to help guide designers during those processes. As a result, some additional language constructs are needed to represent the space of potential solutions, how a potential solution should be evaluated, and how the exploration problem should be framed. In some of these cases, the meaning of existing constructs can be slightly altered; in other cases, additional constructs are added through the use of SysML's profile mechanism.

1.6.2 Model Libraries & Model Transformations

A significant issue with explicitly modeling the architecture exploration is encoding the significant amount of knowledge needed. Even in MBSE methodologies, the knowledge that is explicitly captured is specifically about a single architecture. Instead, in this investigation the following hypothesis that relates to RQ2 is investigated:

H2: Designers could use modularity and composition along with model transformations to reuse knowledge encoded in models.

An organizational scheme for component-level model libraries and generic model transformations is described in Chapter 4. To demonstrate the composition process, generic model transformations are then used to compose these component-level models into system-level analyses. To support the hypothesis, the generic transformations are used to generate both mathematical programming optimizations (in Chapter 6 & 7) and differential equation-based dynamic behavior models (in Chapter 4). The conditions necessary to compose models through model transformations are considered in Chapter 4. It is important to explicitly capture the relationships between models and also include meta-data so that that the appropriate models can be identified and composed.

When representing the problem and constructing the transformations, the enabling characteristic is the commonality in structure of different system architectures and also different types of system architectures. Although the examples in this investigation are from the fluid power domain, some discussion on the commonality between different system architecture's structure is presented in Chapter 3 & 4.

1.6.3 Mixed-Integer Linear Programming

Once an architecture exploration is defined, the appropriate analyses need to be applied to guide the decision making process. In many previous approaches, because of the nature of the solution space genetic algorithms or similar techniques are used to search the space (Koza, 2010).

Instead, in this investigation the following hypothesis related to RQ3 is studied:

H3. Designers could use mathematical programming optimization tools to identify promising solutions early in the exploration. Mixed-Integer Linear Programming should be used for architecture selection.

Mathematical programming is not commonly used for this application, although as mentioned earlier some prior work has established that it is relevant (Biegler, 1997). The current drawback of mathematical programming tools is the difficulty in manually creating the large mathematical programming formulations necessary to represent architecture exploration problems. In this investigation, the transformation approach described in the previous section will be used to automatically generate the text-based models necessary.

This allows further investigation into using mathematical programming in this domain. In Chapter 5, a representation of an architecture selection decision as a mathematical programming problem is represented along with a qualitative comparison of mathematical programming with other potential search approaches. This includes a novel model transformation that includes simplification of certain aspects of the problem definition so they can be represented in the mathematical programming formalism. The underlying assumptions of this mapping are addressed in Chapter 6 to establish that the resulting mathematical programming formulation matches the problem definition. The hypothesis is further supported with examples using the approach to support the design of an excavator's actuation subsystem is described in Chapter 7.

The difficulty in performing an architecture exploration derives from the large variability in a potential system and the large number of potential architectures that could

be explored. Confounding this problem is that these architectures exist in a discrete space.

Where others have attempted to describe this discrete space through generative grammars (Schmidt, 1997, Schmidt, 1998), the approach here is to define this space using constraints. This sort of problem can be described as Boolean satisfaction problem (Creignou, 2001) or a weighted MAXSAT problem (Domingos, 2008b), but the size of the optimization problem and the desirability to allow the representation of continuous variables drives the selection of mixed-integer programming from the mathematical programming domain (Williams, 1999).

To represent mathematical programming problems, the AIMMS modeling system is utilized (Bisschop, 2006). There are a number of similar systems such as the General Algebraic Modeling System (GAMS) (Brooke, 1998) or A Mathematical Programming Language (AMPL) (Fourer, 1990a). These systems provide a textual language for representing mathematical programming problems and a supporting toolset for applying solvers to these definitions. To solve mixed-integer linear programming problems, IBM's CPLEX (International Business Machines Corp, 2009) is used. Using only linear equations to describe a designer's knowledge about component and system behavior is limiting, but at early stages of the process it is crucial to make appropriate simplifications so that the solver is able to handle the scale of the problem.

Although not thoroughly addressed in this investigation and left as future work, in the following design stages, the same problem formulation along with the knowledge gained during this initial step could be used to generate a more accurate mixed-integer

nonlinear programming optimization or an optimization involving more accurate simulations which would search a reduced design space.

1.6.4 Managing Complexity and Supporting Scalability

In order for an approach to be applicable to real-world problems, it needs to be able to scale. This is a drawback with many current computational synthesis approaches, because they rely on a single kind of analysis or level of abstraction. Therefore, in this investigation, one of the central issues is how the approach can be scaled to real world examples. Even in a simplified form, the excavator example provides a good case study for the approach in terms of how well it will scale.

To manage the complexity of a given problem, two separate approaches are considered in this investigation, but they are applied only as necessary in the example problems.

The first is to allow the designer to easily restrict the space of considered solutions so that the search process is only focused on important aspects. For instance, in the excavator example the configuration of the valves could be fixed and the focus could on selecting the appropriate number and configuration for the pumps. This example will be provided along with the main excavator in Chapter 7.

The second is to break down the search process into multiple steps where more accurate analyses are used as the process progresses and the space of solutions is reduced. Although this approach is not highlighted in the examples, how multiple analyses can be created is discussed in Chapter 4 and 6. This is practical in this approach because the same architecture selection problem definition can be used in conjunction with different model transformations to generate analyses with different accuracies. There are multiple

ways in which the knowledge captured in the problem definition can be recombined and each of these recombinations results in a different set of analyses. A conceptual overview of such an approach is shown in Figure 1.1 where the same problem definition is used as basis for the creation of multiple analyses. In this example, three types of analyses are considered with each type focused on a different stage of the exploration process. In this investigation, the focus is on demonstrating mixed-integer linear programming can be used to synthesize potential architectures by exploring the solution space and combining different components into meaningful configurations as described in the previous section. To more accurately size promising architecture, mixed-integer nonlinear programming could be used (Åkesson, 2010a, Shah, 2010c). This allows the inclusion of more complex equations that can provide more accurate predictions about the optimal sizing of a candidate architecture. The final type of analysis could be based on differential algebraic equations that can be used to truly model the dynamics of a system as is consistent with many current optimization approaches. Demonstrating not only that these approaches can be applied but how they should be applied to manage the complexity found in large problems is left for future work.

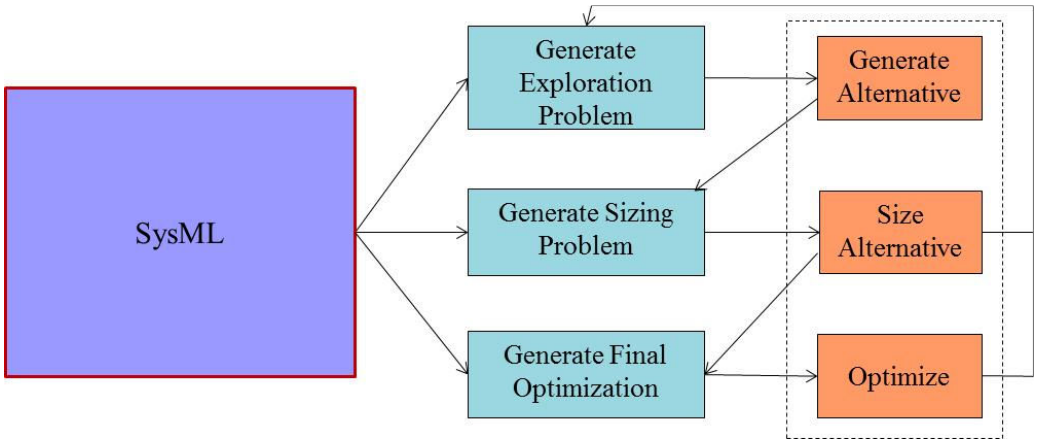

Figure 1.1: Conceptual overview of a multi-staged solution approach.

1.7 Expected Contributions

During the investigation of the research questions, it is expected that a novel framework for solving architecture exploration problems will be developed. A new domain-specific language will be developed to provide systems engineers with the means to represent architecture exploration problems explicitly in a compact and flexible fashion. To simplify the definition of the problem, the use of reusable model fragments will be explored. These fragments will be stored in model libraries along with explicit representations of the relationships between the fragments. The goal of developing this framework is to demonstrate the value that can be added to a design process by explicitly representing an architecture exploration problem with the overall goal of facilitating more rational decision making during the design of a system. By explicitly representing the problem and using model transformations to automatically create analyses needed to solve the problem, the goal is to demonstrate that using information models during the design process can allow for new decision-support capabilities, not just the documentation of a systems engineering process.

Another expected contribution of this work is to demonstrate the creation of analysis models through composition for the architecture exploration process. To allow the representation of the knowledge in the architecture exploration problem as a mathematical programming optimization problem, a composable formulation for this knowledge will be developed. This will also facilitate the definition of a transformation from the architecture exploration problems captured in the domain-specific language into the mathematical programming formulation. In current systems engineering practice and in the previous work in computational design synthesis (that will be discussed in Section

2.3), the creation of analysis models is a manual exercise that often needs to be repeated for each architecture under consideration. The composition approach developed during this investigation would facilitate the automation of this process and could significantly decrease the time and resources needed by designers and domain experts to create these analysis models.

An implementation of this transformation will be created from the domain-specific language into the mathematical programming representation. This will allow the automatic generation of mathematical programming optimizations from the more compact representation of an architecture exploration problem. This transformation will demonstrate a composition approach to automatically create system-level analysis models. Example problems generated using the transformation will allow the testing of mathematical programming solvers, specifically IBM's CPLEX, to demonstrate that mathematical programming techniques are applicable to solving this type of problem. In current practice, mathematical programming techniques are rarely used in the systems design domain. This will demonstrate the applicability of these solvers to the systems design domain, with the overall goal to move towards more widespread application of these existing techniques to systems engineering applications. Also, the automatic creation of the mathematical programming optimizations from component-level analysis models that are composed will demonstrate that by using the appropriate abstraction level, large mathematical programming problems can be created efficiently.

1.8 Investigation Roadmap

The rest of the thesis is broken into several focus areas, each one with one or more associated chapters. As an aside, if the reader wants only a cursory description of

the investigation and relevant results, it is recommended to read Chapter 3 from the beginning through Section 3.4.2, Chapter 4 from the beginning through Section 4.3, Chapter 5 starting at Section 5.3 through 5.5, Chapter 7, and finally Chapter 8. This skips the comprehensive presentation of prior and related work in Chapter 2, but each chapter includes relevant related works to establish the context. Also, this skips some intermediate results and discussion presented in the chapters with the goal of moving quickly to the final results.

The first area, of which this chapter is a part, introduces the research and prior art in this field. These chapters can be summarized as follows:

- Chapter 1 is a high-level introduction to architecture exploration, describing current difficulties and the aims of this research. It contains the overall vision for how designers can model and perform these explorations by representing architecture selection decisions in information models and transforming these models into a mixed-integer linear programming formulation where mathematical programming optimization tools can be used to perform the exploration. This chapter also includes the key research questions and how they will be addressed.
- Chapter 2 is a presentation of the problem background in greater detail. The current state of the art in systems engineering is described to identify how this method fits into the broader field. This includes a review on prior work with system and (to a less-extent) software architectures and architecture design. Also, previous approaches for performing computational design synthesis, a field in which this investigation also fits, are discussed along with their limitations.

The second area shifts to the problem of capturing design knowledge in various formulations and transforming between those formulations. This is presented to enable the exploration of the research questions. This focus consists of four chapters:

- Chapter 3 provides a description of how the designers' knowledge can be formulated into an architecture selection decision and captured within information models. The language used to define these information models is based on SysML, with SysML's profile mechanism used to extend the language where necessary. The foundation for representing an architecture selection decision in a form that is consistent with decision theory is also presented.
- Chapter 4 is a discussion focused on model reuse and composition. The goal is to describe how relevant domain knowledge can be captured in reusable component-level model fragments that are organized into model libraries. To illustrate that these model fragments can then be composed, a transformation is presented and demonstrated from an architecture selection decision where the architecture is known into a dynamic analysis of that architecture.
- Chapter 5 is a description of the formulation of an architecture selection decision as a mixed-integer linear programming optimization problem. Mathematical programming is also compared to other potential search approaches. In order to make the definition of the problem applicable to the composition approach, it is presented in a modular fashion where each element of the architecture selection decision is mapped into a single construct or set of constructs from the mathematical programming domain.

- Chapter 6 is a description of the transformation from the model-based problem definition described in Chapter 3 into the mathematical programming form described in Chapter 5. This transformation enables the practical representation of architecture selection decisions in a form that is consistent with the representation in Chapter 5.

The next area focuses on illustrative examples that support the hypotheses:

- Chapter 7 is a presentation of an engineering example where architecture exploration is performed to select the hydraulic subsystem for an excavator. This problem demonstrates the applicability and scalability of this solution approach when dealing with real-world type problems.

The final chapter brings closure to the research:

- Chapter 8 is a description of the contributions and limitations of the research, and also includes the open questions raised during this investigation.

Also, two appendices of interest are included:

- Appendix A is a compilation of the domain knowledge included in the SysML model libraries.
- Appendix B contains sample AIMMS code generated by the transformation presented in Chapter 6.

CHAPTER 2:

PRIOR AND RELATED WORK

This chapter is a review of prior and related work in systems engineering and computational design synthesis. An examination of the limitations of current practice in systems engineering as it relates to the context of system architecting is also included along with a review how current state-of-the-art design synthesis approaches fail to address these limitations.

Section 2.1 is a review of current decision making practices in systems engineering including the limitations of current practices. The goal is to establish the limitations of current approaches and build a case for using computational tools to support decision making at the conceptual design stage.

Section 2.2 considers potential analysis and evaluation approaches used and how they relate to the conceptual design stage.

Section 2.3 is a review of current computational design synthesis approaches, along with the limitations of these approaches and why they are currently not capable of supporting designers' selection of an approach systems architecture.

2.1 Current Systems Engineering Practice

2.1.1 Systems Engineering Processes

Systems engineering is an interdisciplinary field focused on the design, maintenance, and operation of complex systems (Sage, 2000a). Systems engineering provides various systematic processes and frameworks that help deal with the complexity of modern systems. Because of the complexity of these systems, systems engineers usually focus on designing the architecture of a system and delegate the design of

individual components to domain experts. Although there is not one universal definition for a system architecture, it usually represents the structure of the system (the physical architecture) and its expected behavior (the functional architecture) (Buede, 2000).

Systems engineering provides several methodologies or frameworks for designing and maintaining a system (Estefan, 2007). These methodologies provide a systematic approach for decomposing the system design problem into simpler sub-problems. There is also a growing trend in these methodologies towards model-based approaches and the use of models throughout a systems engineering problem. One such approach is Model-Based Systems Engineering (Fisher, 1998) where models are used instead of documents as the design artifacts used during and resulting from the systems engineering process. One of the benefits of this trend toward formal modeling is the emergence of well-defined modeling languages such as the Systems Modeling Language (SysML) for modeling systems engineering problems (Friedenthal, 2008). By leveraging these modeling languages for representing the knowledge needed to synthesize and analyze architectures, the hope is that these representations will be more intuitive to systems engineers and also allow leveraging other tools that rely on these languages. The current limitation of the systems engineering languages is they fail to provide features for capturing spaces of alternatives.

Since this research is focused on supporting designer decision-making, namely helping designers make ”good decisions,” it is important to understand the characteristics of a “good decision” before delving into systems engineering and systems engineering practice. Many in the design community recognize decision making as a central aspect of

engineering design (Bras, 1993, Hazelrigg, 1998, Olewnik, 2006, Thurston, 1991) and this recognition is spreading into systems engineering and systems design (Parnell, 2011).

A decision is normally defined as an irreversible allocation of resources. Strictly speaking, it is impossible to "un-make" a decision, one may make a subsequent decision to reverse the effect of a previous decision. It is difficult to characterize the goodness of a decision, for decisions are made in the present but result in outcomes in the future (Hazelrigg, 2012) These outcomes are affected not only by the decision but by uncertain events (since the future cannot be perfectly predicted, there is uncertainty in the decision making process). This means that rational decisions may result in bad outcomes while less rigorous decision making approaches may actually result in good outcomes. Therefore, judging the decision purely on outcome is not the desired approach. The best that designers can hope for is to make rational decisions, decisions that are consistent with the designer's beliefs and preferences.

Von Neumann and Morgenstern proved that any decision maker whose behavior is consistent with the axioms of rationality has a real-valued utility function that is such that the behavior of the decision maker can be explained as maximizing the expected value of this utility function (von Neumann, 1980). Such utility functions thus provide a mathematical formalism for expressing rational behavior and a designer should strive to maximize this expected utility. Although there are some challenges to utility theory, most revolve around whether a decision maker is truly rational (Hazelrigg, 2012). There is a growing consensus that the only proper way to formulate the objective of a systems design problem is to use utility theory where the utility function is an expression of a designer's or firm's preference with regard to profit (Castagne, 2009, Hazelrigg, 2012)}.

With this objective defined, the designer should then go about ideating potential solutions, computing this objective for those solutions, and then selecting the best solution. The reality is, in systems engineering and systems design, this is rarely done.

As originally described in Chapter 1, systems engineering design processes use a top-down hierarchical decomposition approach to make decisions which has a number of distinct steps:

1. Identification of the performance objectives and requirements: During this step, the various stakeholders involved with the system come to a consensus of how the system should perform, what are the desired functions, and so on.
2. Preliminary Design: Designers focus on making high-level system selection decisions, such as the selection of the architecture. This phase is often broken down into multiple steps including the definition of a logical (sometimes referred to as a platform independent) architecture and then the synthesis of the actual physical architecture based on this logical architecture.
3. Detailed Design: Here, the focus is on the design of individual system components, which also includes writing the necessary software to control the system.
4. Integration of components into the system: After the appropriate components are selected, they need to be integrated together into a completed system.
5. Testing of the final system: The system is tested thoroughly to insure it is capable of meeting the requirements prescribed in the first step. Issues identified in this step are usually solved in an ad hoc which the goal being to make the system

work. This process can be very time-consuming and the systems deficiencies that are identified during this step are costly and difficult to fix.

Although it is likely that the overall structure of the process can improve, this structure is the result of significant real-world application and testing. In this investigation, the focus is not on changing the structure of the overall process, but instead on improving the results of common tasks undertaken within the context of this process.

To summarize the relevant steps, the design of the system architecture is usually accomplished by starting at stakeholder concerns. These concerns are then transformed into a set of requirements. An architecture that is capable of satisfying the requirements is then created and that architecture is carried forward into subsequent steps. This set of requirements defines the solution space and designers are tasked with creating a candidate solution that is capable of achieving these requirements. There are a number of different approaches for performing the transformation from requirements to a candidate architecture; each company or even design team usually has an ad hoc approach that is favored that relies heavily on designer experience and intuition. In current practice, computational tools are rarely utilized, although the lack of utilization should not be used as the indicator that the process is poor. Usually, ideation techniques are employed that can range from brainstorming to morphology matrices and function-based decomposition. In the ideation phase, one goal is to broaden the space of considered alternatives and often designers are encouraged to consider untraditional solutions. This is often done by providing them external guidance using methods such as bio-inspired design (Parnell, 2011).

Then, some evaluation criteria are established to differentiate the potential solutions. This evaluation criterion may be quantitatively stated as a mathematical equation, as in utility theory or value-driven engineering (Castagne, 2009), or stated qualitatively as is common in Pugh Matrices (Pugh, 1990), Quality Function Deployment (Akao, 2004), and others. These qualitative approaches are the most commonly used when pairing down potential alternatives. They are driven by reaching consensus among a large number of experts, each of which has a large amount of domain knowledge about also personal biases and potentially incorrect preconceptions.

Because the design of systems is complex, often involving numerous stakeholders and design engineers, much effort is taken to decompose the design problem into a set of simpler problems. This is often accomplished using ad hoc approaches where the design of the system begins at a high-level of abstraction and detail is slowly added. During such a process, designers begin at high-level stakeholder concerns and slowly decompose the problem into a growing set of requirements. In the same way, the system is specified by adding detail; as new components or subsystems are included in the system specification, new requirements are also added. Sometimes this is done in a multi-stage process where a logical or platform independent architecture is created based on the requirements, and then this architecture is used as a guideline for the creation of the actual candidate architecture (Friedenthal, 2008). This is similar to function-based design, where the goal is to delay the system specification and reduce designer biases about a particular component technology.

The use of requirements allows certain aspects of the design process to be delegated; the subsystem under design needs to meet the prescribed requirements. These

requirements are often written as shall statements, i.e. “the system shall contain a power subsystem” or “the system shall have a mass less than 500 kg.”

There are numerous issues with this approach, from the practical difficulties of managing and verifying a potentially large number of requirements (it is not uncommon for modern systems to have tens of thousands of such requirements) to the mathematical soundness of decomposing the problem in this manner. The decisions being made by designers during this process are embedded in the resulting requirements, and are difficult to review. Early in the process these decisions are made using qualitative approaches such as those previous mentioned. Even when value-driven or quantitative metrics are established for evaluating a design, these are not employed until late in the process when the architecture and many of the component choices have already been made. There are several reasons for this, including the difficulty of analyzing such complex systems at early design stages and the cost of creating necessary analyses for a number of different architectures. The quantitative trade-studies or optimizations used focus more on a small number of parameters because formulating such optimizations is significantly simpler. Usually, such optimizations already implicitly include the entire structure of the system, and therefore cannot be used during early conceptual design stages.

Recent work in systems engineering, even work in Model-Based Systems Engineering, has focused mainly on the practical concerns (Estefan, 2007, Hazelrigg, 2012, Parnell, 2011, Sage, 2000a). There is limited literature in the systems engineering domain on the decision making process (Parnell, 2011).Value-driven design, where the goal is to design the system to maximize a system value (such as organizational profit or

tactical effectiveness), has called into question the use of performance requirements when designing a system (Castagne, 2009). Requirements can be seen as constraining the design space, and the argument is that simply constraining the design space does not lead to better designs. Value-driven design is being applied during the F6 Program (Castagne, 2009).

To facilitate an approach such as value-driven design where system-value is maximized, several significant improvements to the current state of the art are needed: the ability of quantitatively evaluate architectures during early stages of the design process, the ability to generate these quantitative analyses for a large number of candidate architectures, and the ability to use these analyses to search a potentially large space of promising candidate architectures.

2.1.2 System Architecting

Although the systems engineering discipline encompasses an entire system's life cycle, since the focus of this investigation is on supporting decision making when choosing between multiple architectures it is important to consider the related literature in system architecting. System architecting can be defined as the art and science of designing and building systems (Maier, 2000). In current practice, system architecting is considered a qualitative and inductive process that is performed very early in the design cycle. Most often, only very abstract models are used during this phase, with much of the decision making based on experience and simple heuristics, such as reducing the number of components or connections or reducing coupling. Part of the process is only in identifying

There are a number of frameworks for describing systems architectures, a comprehensive review can be found in (Emery, 2009). One major framework is the ISO/IEC 42010 standard (ISO/IEC, 2007), in which a system architecture is defined as containing the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, as well as the principles guiding its design. This definition is consistent with the definition provided in Chapter 1, although the definition in Chapter 1 is slightly more restrictive because it defines components as having well-defined interfaces by which they are connected.

One of the major goals during this early phase of system architecting is identifying the appropriate stakeholders and their relevant concerns in a qualitative sense; after refinement these are transformed into quantitative descriptions of the objectives for the system. Also during this phase, there is negotiation with the stakeholders to identify a feasible set of these concerns based on available technology. During this phase, experience and various heuristics are employed to identify the potential components and architecture implementations. Very little quantitative analysis is performed because high fidelity models are not available (Maier, 2000).

The goal of this investigation is not to replace the human effort expended in identifying the objectives for the system or in scoping potential implementations. Instead, the goal is to facilitate more quantitative decision making during this process. By providing designers with tools to quickly perform trade-studies and identify promising or infeasible solutions early in the design process, more quantitative decisions can be made when choosing the overall structure of the architecture.

The architecture representation described in the ISO/IEC 42010 standard organizes models based on several views; these views correspond to concerns coming from many different stakeholders (such as engineers, designers, architectures, and so forth). Each of these views conforms to a specific viewpoint which defines the elements that can appear in that view. The description of the architecture selection decision presented in Chapter 3 is informed by this standard, but the description provided there is only a subset of the information captured in the 42010 standard.

2.1.3 Software Architecting

In addition to the field of system architecting, these is also extensive prior work in computer science focused on software architecting. In software design, the goal is also to construct code that is functional, efficient, and easy to maintain. Many of the strategies used in system architecting originate from software architecting and object-oriented programming, such as partitioning and encapsulation. A more complete review of these different approaches can be found in (Gamma, 1995). Also, in the review of computational synthesis approaches in Section 2.3, approaches geared toward the generation of software architectures are also considered.

2.2 Evaluation and Decision Making

There is significant prior art focused on evaluating incomplete designs during stages of the design process which is applicable to the evaluation of candidate architectures. Many of these methods focus on making system-level design decisions and how to model these decisions at the system-level. Before discussing the quantitative approaches, the next section provides a review of qualitative approaches and their inherent limitations. In Chapter 1, the argument was made that the appropriate way to

evaluate an architecture is to choose the best instance of the architecture and evaluate the best instance; this approach will be contrasted with those found in previous work.

2.2.1 Qualitative approaches

Although a primary concern of this research is to apply designer's knowledge captured in quantitative models to support system-level decision making, quantitative models are not the only approach that could be taken. Capturing knowledge in quantitative modes can be difficult and time consuming, so it is necessary to establish conditions when this is desirable. There is the potential to use mental models if designers feel confident enough with their ability to assess the outcomes of their decision alternatives; for example, formulating the problem using utility theory and then using tacit understanding to assign each alternative with an approach value.

The design literature also contains several other selection methods which are commonly employed to make sure of design expertise to evaluate alternatives that are not based on utility-theory. This includes Pugh selection (Pugh, 1991), Quality Function Deployment (Akao, 2004), various rating matrix approaches and the analytic hierarchy process (Saaty, 1990). Although these methods have well-defined and easy to follow approaches for implementation (they are prescriptive and not normative), there is significant doubt as to whether these approaches lead to the selection of the most preferred alternatives. As mentioned previously, these approaches are also the most common in systems engineering, largely out of necessity. This illustrates a clear gap between current best practice and normative theory.

By relying on qualitative mental models, an architecture exploration would also require significant human input. Even if computational support is provided to generate a

large number of potential solutions, designers would need to use their expertise to evaluate this solution population, and possibly evaluate a number of poor solutions.

Another approach is to use computational qualitative reasoning (Bobrow, 1984, Hunt, 1993). In qualitative reasoning, instead of quantitatively solving equations, qualitative statements are made (for instance, the sign of certain terms (either positive or negative) or relative magnitudes). This has the advantage of not requiring exact data or exact models during the reasoning process and can also significantly speed up analysis. The EDISON system is a qualitative reasoning tool to support design improvisation (Hodges, 1992), it uses a qualitative reasoning framework to support the synthesis of machine primitives into simple mechanical systems. A significant issue was the inclusion of appropriate qualitative constructs and machine primitives to allow the system to correctly infer the behavior of a particular system. This is true of any logical system, where a major shortcoming is forcing designers to describe their knowledge about the system in logical statements. The construction of these logical statements is difficult if the goal is for them to describe a systems performance.

2.2.2 Sizing techniques

Another common approach used in systems design is to avoid optimizing for the best instance of a candidate architecture and instead to apply a sizing procedure to choose the appropriate sizing parameters. The engineering product literature contains several examples of these so-called component sizing procedures. A sizing procedure is a sequence of computational steps through which a designer can identify the appropriate component sizes (and often the appropriate component model-number) for a candidate architecture. Essentially, this simplifies the resource allocation or sizing decision by providing fixed mathematical relationships for the sizes; the drawback is these

mathematical relationships are based on a large number of simplifying assumptions which may not hold in every context. It also prevents desires from performing tradeoffs between such factors as performance and cost.

One can find procedures such as these in the literature associated with many domains. Parker Hannifin publishes a guide for how to size pumps and other hydraulic components (Parker, 2002). Given assumptions about the engine, loading characteristics, gearing and design requirements (e.g., lifetime), they define a procedure for determining the suitable pumps and motors. Eaton and Sauer-Danfoss, competing companies, also publish a similar documents for their hydraulic pumps and motors (Eaton, 1998, Sauer-Sunstrand, 1997). This reveals another limitation of sizing procedures, that they are tailored to the context of one product line or one company's products.

2.2.3 A Foundation for Modeling Architecture Explorations

The mechanical design literature provides a number of quantitative frameworks for decision making during the design process. Decision making and optimization methods are also closely linked in the design literature.

One such approach is decision-based design, where decision-making is the central activity being performed during the design process. This approach is normative and prescribes that decision makers should formulate and solve decision problems in a mathematically consistent and rational manner. The limitation of normative frameworks is they describe how designers *should* make decisions; they rarely tackle the practical details of how designers *can* make decisions.

Then, the task is to formulate the appropriate decisions and organize the relevant information. One such formulation technique is the Decision Support Problem Technique in which the design problem is formulated as a Decision Support Problem (DSP). A DSP

is a template for structuring various types of decision problems. The one of most interest to this work is the compromise DSP (cDSP) (Bras, 1993, Karandikar, 1989).

Although these frameworks provide a rich (textual) language for designers to represent their problems, they lack a clear approach to defining the system alternatives. Defining the decision in a form that is convenient to designers is a major focus of this investigation; whereas previous approaches have established mathematical representations for the decision problem, in this work a domain-specific language is defined to allow designers to conveniently capture the relevant knowledge. Also, the optimizations are based on low-level system variables which relate to the physical construction of a system. When these approaches are demonstrated in prior work, the selection of the architecture has already occurred.

2.2.4 Surrogate Models

A difficulty in evaluating potential architectures is the computational expensive of running the necessary analysis models. To alleviate this issue, another commonly used approach is to replace these expensive computational models with surrogate models (sometimes also called a meta-model or a reduced order-model). One achieves this by constructing a more complex model, sampling this model, and then fitting a simpler model to the data. There are several example of surrogate modeling, such as Bayesian techniques, Radial-Basis Functions, splines and Kriging models.

Kriging models have their origins in geo-statistical applications that involve spatially and temporally correlated data (Matheron, 1963). Because a kriging model is an interpolation model, it fits all given data points exactly. Kriging models were used in the context of the optimization of a hydraulic excavator in (Conigliaro, 2009) and by Simpson to estimate subsystem uncertainties (Gano, 2006).

There are other interpolation methods, such as radial basis (Dyn, 1995) or splines (Friedman, 1991) which could also be used in these applications. There exist several surveys of surrogate modeling techniques being applied in a design context (Jin, 2003, Simpson, 2001, Wang, 2007).

The difficulty in using surrogate models is they rely on input-output data from existing models; these models still must be constructed by designers. Also, surrogate models often assume that the model being approximated is continuous; this is not the case when considering multiple candidate architectures, therefore a new surrogate model is needed for every candidate architecture.

2.2.5 Predictive Models

Instead of attempting to evaluate an architecture using low-level component parameters such as bore diameter or gear ratio, another approach is to predict the architectures performance based only on higher-level parameters such as mass or cost (Malak, 2010). To accomplish this, Pareto dominance analysis is used to eliminate inferior components and then a surrogate model is fit to the remaining data points. This surrogate model describes a relationship not between low-level component attributes but instead items such as cost or mass. In this investigation, this concept is used to simplify the sizing approach by considering mainly higher-level parameters.

2.2.6 Optimization in Systems Design

There is extensive prior art in applying optimization in the context of design, but often optimization is applied to a particular subsystem or after a significant portion of the design is already fixed. Of particular interest to this investigation is optimization methods focused on system-wide optimization. There are two major frameworks which fit into system-wide optimization, collaborative optimization, where a system-level optimization

coordinates several component-level optimizations, and multidisciplinary design optimization (MDO) (Sobieszczanski-Sobieski, 1997) where the system analysis is decomposed by various disciplines (static, dynamics, thermal, acoustics, and so forth) instead of components. One top-level optimizer is used to manage multiple sub-optimizers which are optimizing across different disciplines or components. The top-level optimizer guides the sub-optimizers and also insures that they are operating on a consistent description of the current solution. The issue with these optimization methods is that they are designed to occur after the system architecture has already been fixed.

2.3 Computational Design Synthesis

A key issue with approaches from mechanical design is they rely heavily on the expertise of human designers during the architecture selection stage. On the other hand, there is significant work in the research community on computational design synthesis, where computational tools are used to transform the definition of a design problem (often stated as a set of requirements) into potential solutions. For many computational design synthesis approaches, the focus is directly on synthesizing potentially promising candidate architectures (Agarwal, 1999, Cagan, 2005, Helms, 2009, Yeomans, 1999), although rarely do design synthesis methods differentiate between synthesizing the architecture and sizing this architecture.

The advantage of computational design synthesis methods revolves around the promise that they can explore a wider range of solutions than human designers and they can also reduce the tedium of some design tasks leaving designers more time to perform other, more creative, activities.

These supporting arguments are similar to those made in the first chapter. In addition, computational tools should not be thought of only for supporting the exploration

of additional solution or the reducing of tedious tasks, but also for supporting a designer's decision making process in choosing the appropriate architecture.

Previous work on design synthesis usually focuses on a particular domain, such as gears (Starling), shapes (Agarwal, 1999), or chemical networks (Biegler, 1997). Systems engineering problems are by definition interdisciplinary, so it is important to consider how to represent the problem of architecture synthesis in a form where knowledge from various disciplines can be easily incorporated. Some approaches represent alternatives by having designers enumerate every combination which for most systems would be intractable because of a combinatorial explosion of possible combinations. Other approaches use custom code to create the data structures used to express possible design solutions (Agarwal, 1999). Because of the wide range of possible domains in systems engineering, it is important to consider a more general approach where designers can express their knowledge in flexible data structures that are more consistent with the shift toward model-based approaches.

2.3.1 Function-Based Approaches

In early stages of a systems design process, designers often elicit and decompose the functions of a system. The end result is a functional architecture; designers use this functional architecture as a basis for designing the physical architecture by choosing structural components that embody the appropriate functions. Function-based approaches attempt to support this process with computational tools by creating repositories of common functions, their inputs and outputs, and compatibility between functions and then use these repositories as a basis for automatically generating functional architectures.

Designers represent their knowledge about potential functions and also their embodiments within a repository (Bohm, 2008, Bryant, 2005). Designers provide a description of the functions of the system, and the tool synthesizes functions from the repository until the functional architecture is able to achieve these functions. Compatibility between functions is then often used to prune infeasible results. Functions are represented as transforming inputs into outputs; they are also usually atomic building blocks of systems, for example elements that transform energy.

The significantly limiting factor of these frameworks is they focus only on the functional representation; assuming that a functional representation is the appropriate abstraction when designing a system is a very limiting assumption.

Also, the embodiment design (creating the physical architecture) must still be accomplished by human designers, and this includes sizing the components. Although identifying systems that are functional is important, it does not help designers choose between these system because designers must still use traditional methods to choose between these functional designs.

2.3.2 Grammatical Approaches

Various methods are presented in the literature for using design grammars to provide automated synthesis to explore the design space of a particular problem. The design grammars are defined through the use of graph transformations. Just like the English grammar specifies which sentences are allowed, a design grammar specifies how design alternatives can be structured. Design grammars have been commonly used in building design (Stiny, 1980), software engineering (Agrawal, 2002, Le Metayer, 1998) and engineering design (Alber, 2002, Baker, 1990, Campbell, 2003, Haq, 2005, Heisserman, 1994, Mullins, 1991b, Rinderle, 1991, Schmidt, 1996). Although in most of

the work in engineering design the emphasis has been on geometry design rather than systems structure, systems design in terms of configuration design is addressed in (Schmidt, 1997, Schmidt, 1998) and specifically for the design of hydraulic systems in (da Silva, 1998, da Silva, 2000, Westman, 1987).

Many of these methods use graph grammars (Nagl, 1979) as the formalism for representing a space of possible design alternatives. Often, each alternative is represented using a graph where nodes represent specific components and edges represent connections between those components. Graph transformations are then used to generate new alternatives by rewriting existing graphs that represent either completely or partially specified alternatives. Graph grammars provide a formal language for specifying the design space (Mullins, 1991a), but often this design space is specified in an ad hoc manner. Graph grammars have been successfully used in a number of applications, but the transformations can be difficult to define (Bolognini, 2007, Starling, 2005). Although the data structures and transformations approaches can be based on custom code, computer-aided software engineering tools have been used recently to simplify their specification (Fischer, 1998, Königs, 2006). Also, grammars are usually only used for creating topologies and a completely separate approach is used to solve for component parameters. One notable exception is attribute grammars (Mullins, 1991a) where configuration and parametric design is considered a part of the grammar. The drawback of using graph transformations is that there is a need for a large number of transformations that must be specified manually. Insuring that the result of a transformation is still within the space of alternatives is also difficult and requires the transformations to be defined very precisely. These limitations make it difficult for

designers to encode their knowledge within such transformations. Also, analysis knowledge cannot be included in the transformation process; instead the alternatives themselves need to be analyzed in a separate step.

2.3.3 Constraint-Based Approaches

Another set of methods can be classified as constraint-based approaches; usually the constraints are specified as either a set of equations or using a custom constraint language. Mathematical programming approaches have been used for automatically synthesizing chemical reactor networks (Biegler, 1997, Yeomans, 1999). The chemical network is represented as a superstructure; a superstructure is the union of all possible alternatives. It is a conglomeration of all potential architectural options. Decision variables are used to represent which options of a superstructure are included in a particular alternative. Constraints are then added to specify which sets of options specify valid alternatives and to specify the expected behavior of a particular alternative. The constraints are represented as a set of algebraic nonlinear constraints.

These approaches demonstrate the potential for the application of mathematical programming to architecture exploration and also demonstrate the ability to use the same framework both to select and size an architecture. The selection of mixed-integer programming for consideration in this investigation was based largely on this earlier work. In addition, previous approaches have also demonstrates that mathematical programming problems can be represented using object-oriented modeling languages which are then flattened (Åkesson, 2010a). In this prior work by Åkesson, the goal was to optimize the controller of a fixed-system.

In Model-Driven Engineering (MDE) (Chanron, 2006), metamodels and constraint languages are being used for synthesizing the structure of software. A

metamodel in this context is a model that defines the possible entities and relationships that can be used in conforming models. It defines a space of conforming models; although the number models in this space can be infinite. By specifying additional constraints in a constraint language such as the object constraint language (OCL) (Warmer, 2003) or Alloy (Jackson, 2002), constraint-satisfaction approaches (Kumar, 1992) can be used to generate software alternatives that both conform to the metamodel and satisfy the constraints (Saxena, 2010).

Others have found that designers can learn to use constraints to define a design space, and that only a small number of different constraint types are needed when defining such a space (Wyatt, 2012). The current limitation is that the constraints being used are hard constraints on the systems structure which can make it very easy to over constrain the design space and requires iteration by designers to correctly specify the space. This results in designers to representing their knowledge, running a search process, analyzing the results, and then adjusting their representations which was found to be very time consuming. Since the structure and goals of this approach are similar to the approach in this investigation, how it compares is discussed more thoroughly in Section 7.8 after the presentation of the example problem.

Constraint-based approaches have been used specifically for the automatic design of hydraulic systems. Constraint-satisfaction has been used for choosing the sizing parameters of a system (Leweling, 2000), but the systems architecture was considered to be known a priori.

2.3.4 Adaptation-Based Approaches

Case-based reasoning (Vong, 2002) has also been used to attempt to retrieve hydraulic circuits that satisfy some design requirements. In case-based reasoning, a

catalog of generic cases is created along with the relevant solutions. When a new case is presented to the case-based reasoner, it locates the nearest cases in the catalog and then selects the matching solutions. Case-based reasoning has the disadvantage of needing a large number of cases to produce good solutions for varied cases. Also, there is the underlying assumption that case-based reasoning approaches are incapable of arriving at truly novel solutions because they are based on capturing the characteristics of existing solutions and selecting the appropriate existing solution.

2.3.5 Knowledge Capture

Computational methods can also be categorized by the domain knowledge the method tries to capture. Generally, methods can be grouped into one of two major categories: broad and focused methods. Broad methods are designed to be applied to a wide-variety of problems and attempt to explore a very large space by using very generic knowledge about a domain, for instance mechanical systems. Part of the issue with these broad exploration methods is that although they are suitable to a large class or problems, they do not allow designers to effectively encode their knowledge.

Focused methods on the other hand are designed specifically geared toward solving a small set of problems, and although they can be specifically constructed to be effective on this sort of problem, they often require a large amount of custom code or do not allow designers to easily represent additional knowledge about the domain. Previous work in synthesizing hydraulic systems has investigated using a focused method to perform architecture exploration (Pedersen, 2007). Here, knowledge about the hydraulic domain is encoded in custom analysis models and a multi-level genetic algorithm-based approach is used to search the space. Because of the relevance of this method to this

investigation, how it compares is discussed in Section 7.8 after the presentation of the example problem.

The limitations of these previous approaches stem from a failure to effectively capture and use a designer's knowledge about the solution domain. Others have recognized this problem and recommend the use of formal modeling to capture this domain-specific knowledge (Antonsson, 2001).

These methods still fall short because they do not completely encode the architecture exploration problem; architecture synthesis is considered separately from component sizing, and the analysis knowledge needed to evaluate the architecture is not captured or used within the same framework as synthesis knowledge. Instead, constraints on the architectures topology are applied to constrain the design space until only plausible solutions are generated. Having systems engineers represent their knowledge in rigid structures such as formal grammars (Antonsson, 2001) or structural constraints (Wyatt, 2012) can be difficult, with user studies showing the need for repeated iterations to define an appropriate design space (Wyatt, 2012). Finally, because analysis knowledge is not used during the synthesis step to guide the exploration, these approaches are inefficient. Therefore, they work well for toy examples but are unlike to scale to typical systems design problems.

2.3.6 Searching the Design Space

An efficient search method is needed to explore the space of possible solutions. Searching a design space of complex systems can be both expensive and time-consuming because of the cost of generating a large number of alternatives and the cost of executing detailed simulations to evaluate them. Genetic programming techniques are a very common method for searching the design space when synthesizing alternatives. They

have been shown to generate high quality solutions in a number of fields, for example electric circuit design, mechanical systems, and optical lens systems(Koza, 2010). The mutation and cross-over operations used in genetic algorithms usually modify existing solutions making genetic algorithms a commonly used search technique for grammar-based approaches (Emmerich, 2001).

Others have used agent-based approaches to search the design space (Agarwal, 1999), specifically for simple electromechanical systems (Campbell, 2000). By employing independent computational agents, design synthesis algorithms can be decomposed and distributed across multiple computers. If these agents are considered as models of individual members of the design team, agent-based approaches can be used for design exploration. Agents usually have different roles, such as adding or subtracting components or evaluating an alternative. The drawback of such approaches is that they can be inefficient when searching a large design space because each agent performs elementary operations. Usually these frameworks also rely on custom representations for communicating between agents, making it difficult to incorporate additional features.

2.4 Summary

In current systems engineering practice, the task of creating a system architecture and to a lesser degree designing a system is the role of systems engineers and systems architects. As part of the design process, domain experts are also engaged in more detailed design steps. There is significant prior research into how systems engineers and designers should design systems. In current practice, much of this process is ad hoc with qualitative methods being employed by designers. Others have identified the need for more structured and quantitative processes, specifically in the field of computational

design synthesis where computational tools are used to synthesize potential alternatives. Current tools lack effective formulations of the architecture exploration problem, current formulations usually lack the ability to encode both knowledge of the design space and analysis knowledge to analyze and evaluate alternatives. Because of the nature of the problem, they also often rely on inefficient search methods. This is a gap in current methods, with the need for a more complete problem formulation and more efficient and effective search methods.

CHAPTER 3:

REPRESENTING ARCHITECTURE EXPLORATION PROBLEMS

In this chapter, the focus is on how designer's knowledge can be formulated to describe an architecture exploration problem as an architecture selection decision. The goal is to describe a generic language in which designer knowledge can be encoded; this language is then illustrated with the excavator example in Section 7.1. Describing the structure of the language without providing concrete guidelines to utilize the language often makes it difficult to understand how the language will be used in practice.

The goal of this chapter is to support H1:

H1: Designers can represent their architecture exploration problem in information models as an architecture selection decision consistent with decision theory using a domain-specific language.

The argument supporting the hypothesis is structured in two ways: first an argument is made for describing the exploration problem as an architecture selection decision, and then a domain-specific language is presented to represent architecture selection decisions. The structure of an architecture selection decision is based on two important factors: the structure of a decision from decision theory and the assumption that systems are composed of well-defined components that are connected together by well-defined interfaces. As will be demonstrated, this significantly simplifies how the problem is represented.

Deciding on a system architecture is a non-trivial task, and it usually involves a large number of stakeholders. These stakeholders bring their unique concerns, viewpoints, and also domain knowledge to the problem. This leads to a large amount of

available knowledge that relates to an architecture exploration problem and also a number of considerations that must be taken into account when designing the final system. No one person can internalize all of these considerations and ideate the appropriate architecture. It is essential for these experts to be able to communicate effectively with one another and also for the large amount of available knowledge to be reviewed and applied. Therefore, the problem needs to be defined in a consistent manner that is both sufficiently unambiguous and easy to use.

The overall goal of this research is not to simply provide a better documentation approach for the exploration problem; this in itself does not add much value to the process. Instead, the goal is focused on improving the decision making process at early conceptual design phases, specifically when making architecture-level selection decisions. The first step (described in this Chapter and Chapter 4) is to clearly define these decisions in a formal representation that is computer interpretable; then once the decision is defined, computational tools can be applied to support the decision making process (described in Chapter 5, 6, and 7).

Traditionally, the architecture exploration process itself is not formally documented. Instead, most of the documentation efforts focus on the results of the process, although it is true that sometimes rational is included, this is usually an afterthought. Also, often the system specification is captured through requirements that a system must meet. These requirements are derived by decomposing more abstract requirements. This process for decomposition and its impact on the quality of the final design is not well understood. Since the decomposition process is based mostly on best practices and not on a strong theoretical framework, it is also not standardized and often

occurs in an ad hoc way. From a theoretical point of view, requirements are constraints that reduce the size of the potential solution space. Decomposing from top-level requirements actually reduces the likelihood that designers are able to meet the top-level requirements because it places additional (possibly unnecessary) constraints on the system (Hazelrigg, 2012). Also, requirements provide no means to distinguish between potential solutions that meet the requirements. Therefore, they can only help in picking a good enough system, not the best system. Clearly, this obfuscates whether the selected architecture is truly the best architecture for the particular case. The reason that requirements decomposition is so common in practice is because it is easier to implement than more rigorous approaches; it allows systems engineers to reduce the complexity of designing a system by breaking down the problem and assigning different pieces to different design teams.

An alternative approach that is gaining support is value-driven design (Castagne, 2009). In value-driven design, instead of describing requirements the system should meet, an objective is formulated and the goal is to find the system that maximizes that objective. Often, the objective is something that is easily agreed upon, such as the maximization of profit. Value-driven engineering is based on utility theory and provides a fundamentally sound foundation but is difficult to implement.

In this chapter, an information modeling language is presented for using information models to define an architecture exploration problem in a form that is consistent with decision theory by representing it as an architecture selection decision; this is done to allow designers to use more formal and structured architecture exploration processes. This language allows designers to capture the system alternatives being

considered, evaluation criteria, and the analysis knowledge needed to evaluate each architecture relative to the evaluation criteria. To model an architecture exploration problem, it is necessary to model many aspects of the system along with the associated plethora of domain-specific knowledge and such a model will need to be represented in a sufficiently flexible formalism. Systems engineering provides a foundation because it is an encompassing discipline specifically focused on managing the knowledge associated with a systems engineering process. Therefore, it is natural for this investigation to build on existing systems engineering practices.

Classically, to capture the wide range of knowledge that was needed during a systems engineering process, paper documents would be used. These provided engineers with a very flexible formalism that was accessible to the various stakeholders. The problem with documents was they are difficult to review and maintain, and the cross-cutting dependencies between different viewpoints are difficult to represent.

As a result, there has been a trend in the systems engineering community toward Model-Based Systems Engineering (MBSE) (Fisher, 1998). In MBSE, instead of using paper documents, systems engineers use information models to document their systems engineering processes. As a result of the MBSE trend, the Systems Modeling Language (SysML) has emerged as a general-purpose visual language designed to capture many of the different facets needed to describe a systems engineering problem (Object Management Group, 2006). The difficulty in moving from documents into models is that models are by their nature less accessible to shareholders and also much less flexible. Part of this difficultly can be reduced by using SysML as the basis for this approach because the existing expertise that systems engineers have with SysML can be leveraged

(Karban, 2008). Also, there exist a number of high quality commercial authoring tools for SysML models, making the language more accessible.

However, simply using SysML as the basis is not sufficient. With the growing trend in the systems engineering community toward Model-Based Systems Engineering, others have recognized the need to capture the potential solution space for system architectures. The drawback with most of these methods is the focus on the variability of potential component concepts within a (mostly) fixed system architecture. Using component variability can express whether some components are included or not within the architecture, but how those components connect also has to be captured. This is a significant shortcoming because much of variability potential arises from the ability to connect the same components in unique ways.

The rest of this chapter is outlined as follows. Section 3.1 presents prior and related work focused on the capture of designers' domain knowledge in information models. Section 3.2 discusses the foundation for modeling architecture exploration problems as architecture selection decisions. Section 3.3 describes an architecture selection decision and how modularity can be used to simplify the specification. Section 3.4 describes the language for defining these decisions. Sections 3.5 and 3.6 wrap up this chapter with some further discussion of the method.

3.1 Prior and Related Work in Modeling Designer Knowledge Explicitly

3.1.1 Capturing Variants

Available modeling languages, such as SysML, are currently used to capture concrete artifacts such as a single candidate design. SysML is designed to capture many aspects about a system, from numerous requirements to analyses to behavior and so forth,

although these are captured in relation to a single system; although hierarchy, abstraction, and other techniques are commonly used to model the system, the final result is still a model of only a single system.

This poses a problem for defining an architecture selection decision because instead of modeling a single candidate design, a designer must model a space of potential solutions. Others have also identified the need to extend SysML to allow a space of solutions to be modeled (Trujillo, 2010). Usually, this space of solutions is actually a product platform and the solutions being modeled share many common elements with only some included components varying. These approaches are not sufficient for modeling an architecture selection decision because they are not suited for capturing a multitude of component configurations.

Dauenhauer et al. (Dauenhauer, 2009) motivate the need for variability in automation systems and propose implementing reusable model fragments coupled with model transformations to construct potential solutions. Although the authors provide a high-level overview and motivation for the approach, they do not provide either a concrete language or a reference implementation in which variability can be modeled. This investigation builds on a similar motivation (Kerzhner, 2009) for using model fragments captured in SysML coupled with model transformations.

Also, previous approaches model system variability apart from other aspects. In the approach provided here, the goal is to define a model that includes the relationship between the space of potential solutions and the analysis knowledge needed to evaluate those solutions.

3.1.2 Domain Specific Languages

To address the lack of a language to model system variability, one option is to define a domain-specific language for capturing architecture exploration problems. A domain-specific language (DSL) is a language that is tailored to describe a particular problem domain. The use of DSLs to define the models has the advantage of providing designers, who have expert knowledge about a particular domain, with languages that are not only unambiguous but also easily interpretable. This is not always true of more general languages because they are often more abstract.

There are several approaches to define DSLs (Weisemoller, 2007) but, in general, an abstract and concrete syntax need to be defined. The initial step to defining a DSL is creating a metamodel. A metamodel defines the abstract syntax of a domain specific language; it defines in an abstract way the constructs of the language and their relationships. A metamodel represents the structure of the language independent of any particular representation or encoding. Every model described by the DSL is an instance of the DSL's metamodel; a metamodel describes a model just as a model describes a "real world" element (Fisher, 1998). After the metamodel is defined, the DSL is implemented by defining the concrete syntax. This syntax consists of the textual or graphical constructs with which the modeling is done.

The architecture selection decision DSL is a major part of the research presented in this dissertation. There are several standard ways that DSLs are defined in model-driven software development and other software development processes. (Weisemoller, 2007). OMG has introduced *profiles* as a light-weight mechanism to extend UML. Also, OMG provides the Meta Object Facility (MOF) (Object Management Group, 2007) as a metamodeling language for the definition of domain-specific languages.

When combined with constraint languages, profiles provide extensive expressivity. Also, they are widely supported by current UML tools. Unfortunately, in general constraint languages are difficult to use because there are ambiguities concerning inheritance between stereotypes and high-quality tool support is not available for common constraint languages such as the Object Constraint Language (OCL) (Weisemoller, 2007).

UML can also be extended through the use of a MOF tool and the merge concept from the UML Infrastructure (ISO/IEC, 2005). This allows more expressivity than simply using a UML profile but is not widely supported by UML tools.

Finally, a totally new metamodel can be defined for the DSL using a MOF tool. This has the advantage of being the most expressive and flexible method to defining a DSL. Unfortunately, additional steps need to be taken to implement the concrete syntax of the DSL.

An approach to combining the definition of the metamodel for the DSL with adaption of existing tools to use the DSL is also presented by (Weisemoller, 2007). This approach is illustrated in Figure 3.1. The general steps taken are:

1. The abstract syntax of a DSL is defined in a MOF-compliant metamodeling tool.
2. A UML Profile is sued to define the concrete syntax of the new language with constructs similar to those used by UML.
3. An implementation of Query/View/Transform based on Triple Graph Grammars (Königs, 2006) is used to translate the stereotyped UML model into an instance of the metamodel. More on these transformation approaches can be found in Section 6.1.

This approach has the benefit of being both expressive and quickly implementable to provide tool support.

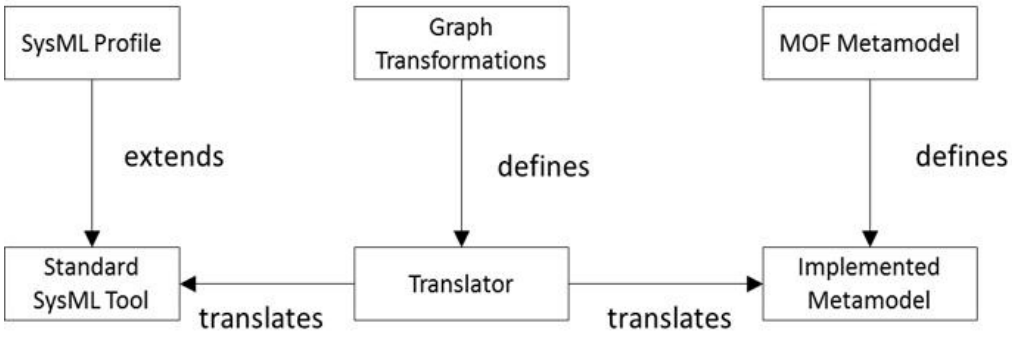

Figure 3.1: A combination of UML profiles and metamodel based technologies

Here, the approach will be to use SysML constructs when possible, and extend SysML using the profile mechanism as needed. This profile along with the SysML profile and underlying UML metamodel comprise the metamodel for this new language. This will define a DSL which is largely based on SysML, leveraging existing experience with SysML and reducing the number of new constructs designers will need to learn.

3.2 Foundation for Modeling Architecture Exploration Problems

Before continuing with the description of the approach, it is important to understand the knowledge that needs to be encoded in an architecture exploration problem. After characterizing the knowledge, requirements are derived to guide the creation of the modeling framework described in the following sections.

Current architecture exploration efforts are ad hoc in nature and rely heavily on designer expertise and intuition. The goal of this work is to improve the rationality of the architecture exploration process by improving the decisions that designers' make. Seeing

the architecture explore process as a chain of decisions, as is common in mechanical design, is necessary to provide a strong theoretical foundation (Donndelinger, 2006).

When the architecture exploration process is seen as a chain of decisions, decision-based design (DBD) can provide a formal, structured, and rational framework for making these decisions (Hazelrigg, 1998 , Thompson, 2010). In DBD, a design problem is broken down into a set of decisions where the formulation of each decision is based on decision theory. When picking an architecture, the designer is making a decision (or a set of decisions) based on his knowledge and beliefs. This decision can be structured using decision theory, where the designer is picking from a set of potential alternatives. For each alternative, the designer is predicting how well the alternative will perform (its outcome) by using his knowledge and beliefs about the alternative. Finally, once the outcome of each alternative is understood, the designer chooses the most preferred outcome using some selection criterion which contain his preferences.

The classic structure for a decision is shown in Figure 3.2. A decision is made among choices A1, A2, A3. Each choice leads to potential outcomes and there is uncertainty involved in the mapping between choices and outcomes. Alternatives can only be evaluated based on their outcomes, with some evaluation metric (utility theory) applied to rank these outcomes. The expectation of the utility of the outcomes is then taken to determine the alternative that is expected to deliver the best result.

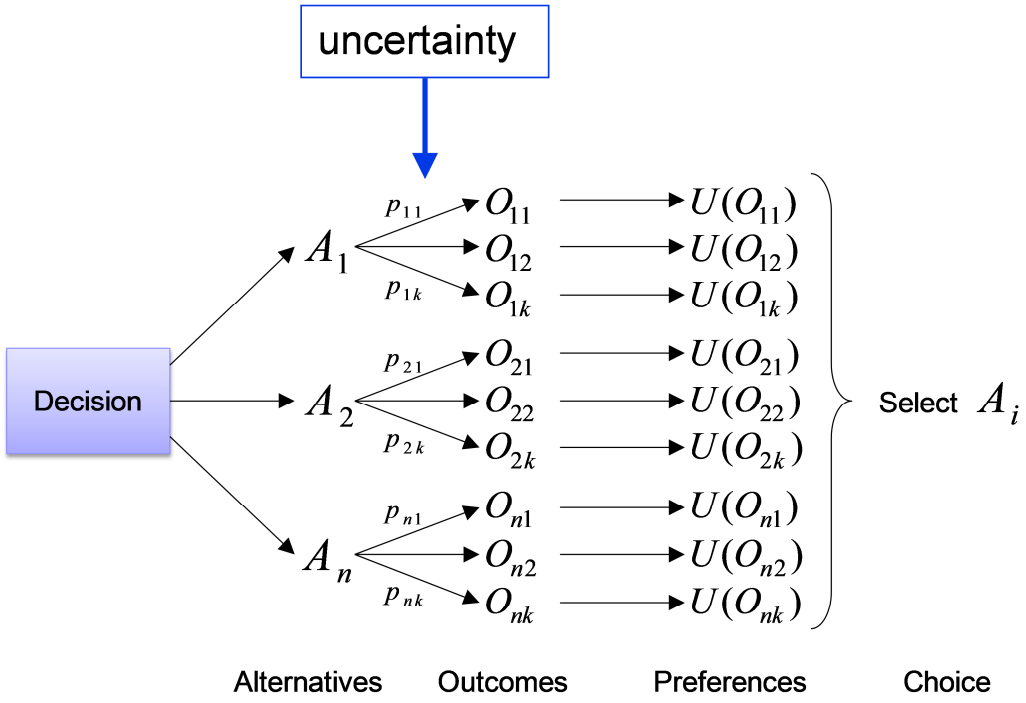

Figure 3.2: Breakdown of a Decision into its basic features.

Using DBD and decision theory as the basis for understanding the architecture exploration process does have several drawbacks. In order for designers to make rational decisions, decision theory stipulates that each decision is made by a single decision maker.

This is completely contrary to current systems engineering practice, where a large number of designers are involved in choosing the architecture. Input is considered from many sources, including human experts, and in current systems engineering practice there is not a consensus of how to bring together these different views. The other difficulty with using decision theory as the basis is that it assumes that a choice is being made over known alternatives. At early conceptual design stages, a significant task of designers is to simply ideate potential architectures.

Considering these drawbacks, decision theory still provides a very strong foundation for what knowledge needs to be captured to formalize an architecture exploration problem. Also, if the architecture exploration problem is represented as a decision, then it can be represented in a form that is consistent with decision theory which is a first step toward improving the rationality of the architecture exploration process. Based on the structure of the decision in Figure 3.2, the model needs to capture the potential alternatives, how well those alternatives perform (their outcomes), and some selection criteria to sort the outcomes and pick the best. A framework for what is modeled and how these relate to each other is illustrated in Figure 3.3. The blue boxes represent elements that are captured while the green oval represents the goals of the optimization process. Simply modeling each of these aspects is not sufficient, in addition any representation has to not only be unambiguous and computer interpretable, but also convenient for a designer. If capturing the designer's knowledge about the problem in this form is exceedingly difficult or time consuming, it will detract for the probability that such a method will be implemented in current practice. In order to efficiently capture the space of potential architectures, a compact formalism is needed to represent a very large space of alternatives.

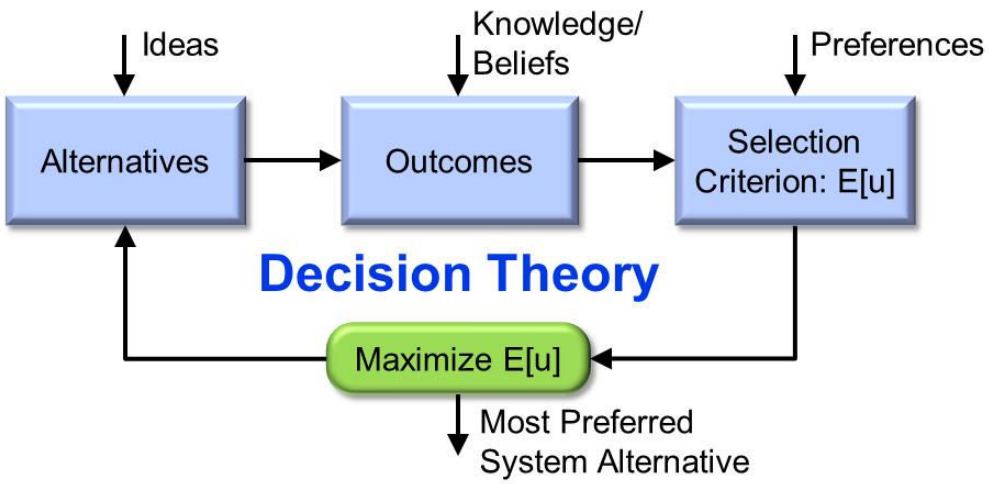

Figure 3.3: Decision Process adapted from Hazelrigg. (Hazelrigg, 2012)

The components that are included in a system may come from a number of different domains, i.e. mechanical, electrical, and so forth. Also, different stakeholder concerns require different system analyses, which also require domain-specific analysis knowledge. Instead of formulating a language that is specifically geared toward one domain in particular, the goal should be to create a representation that is flexible enough to cover the entire range of potential architecture selection decisions. Instead of structuring the representation for a particular domain such as mechanical or thermal systems, the entire domain of systems is considered by identifying commonalities in the structure of a system architecture, regardless of the domain.

3.3 What is an Architecture Selection Decision?

Before continuing, it is important to provide a clear understanding of an architecture selection decision and the simplifying assumptions used in this investigation to allow the representation of an architecture selection decision in a compact and modular fashion.

When considering an architecture selection decision, the alternatives are different system architectures. In classic systems engineering processes, an architecture is selected during the preliminary design phase, and this architecture is carried forward into more detailed design stages where the focus is on sizing the different components. This type of decision breakdown is illustrated in Figure 3.4 as a sequential decision. First, a decision is made between the different architectures, and then once a particular architecture is chosen sizing decisions are made to size particular components. This is represented by the set of arrows following each architecture, with a1, b1, and so forth representing particular sized versions of component types A, B, and C. As presented, this implies the sizing process is one of picking between existing components but nothing about the structure as presented precludes the inclusion of new or custom made components. In order to make a rational decision given this sequential structure and pick the appropriate architecture, DBD specifies that the utility of each leaf (each completely sized architecture) is evaluated and then the best path is chosen (i.e., the selection of the best architecture based on the best sized instance). As mentioned previously, this style of decision making is not used in current practice, instead ad hoc selection criteria are applied to the architecture selection decision, and not until after this decision is made is sizing considered. The other issue is that even representing the architecture selection decision using this tree-like structure where each individual architecture is enumerated is very difficult. For even a small space, asking a designer to explicitly represent each architecture in this fashion would be time consuming and difficult.

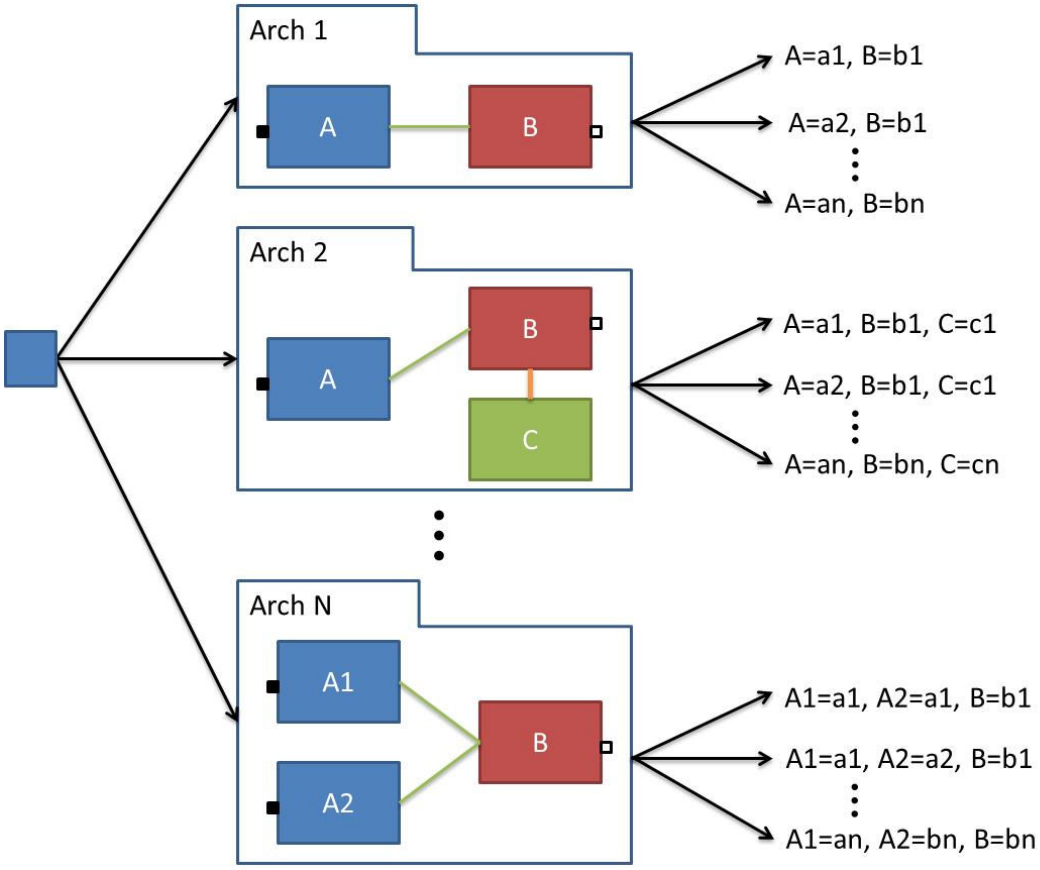

Figure 3.4: Classic description of an architecture selection decision.

By employing the assumption that a system architecture contains well-defined components and subsystems that are connected together into more complex systems, structuring the decision can be simplified because instead of representing each alternative separately, the entire space can be represented as a union of potential components and connections.

This significantly simplifies the representation of the problem because instead of encoding each individual alternative, the problem can be defined modularly with a focus on capturing the structure of the individual components and connections and the composition relationships between them. This is similar to composition approaches for generating analysis models (Kerzhner, 2010, Kerzhner, 2011, Shah, 2010a), but here

instead of generating a single analysis for a particular architecture candidate, the goal is to represent the design space modularly and then if needed generate an analysis (or a set of analyses) that encompasses all possible candidate architectures. A possible modular representation of the space presented in Figure 3.4 is shown in Figure 3.5. Instead of enumerating each architecture, an abstract system is represented as an empty system boundary. This abstract system can then be composed of the different components as restricted by the related multiplicities which are assignments of the potential number of each component.

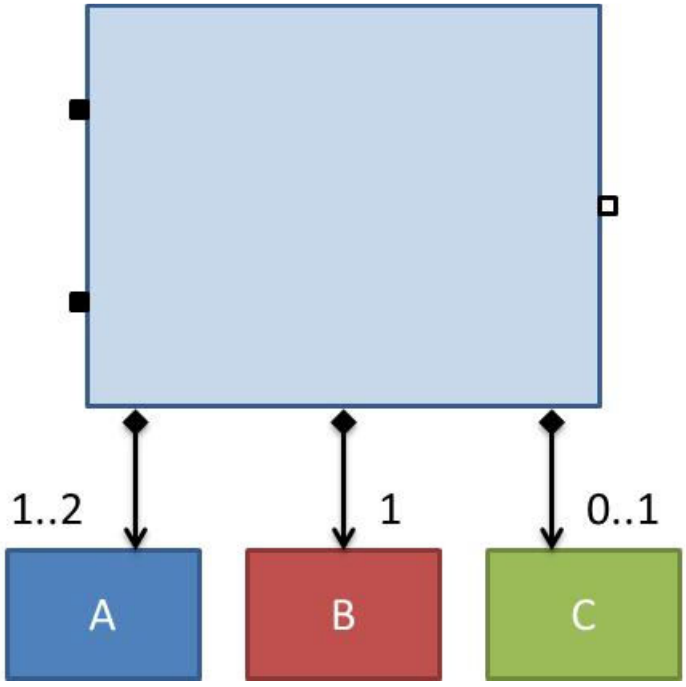

Figure 3.5: Modular representation of the architectures considered in the architecture selection decision.

This representation is sufficient for representing the different components that can be included in the design space, but the connections must also be modeled. These are not represented in the figure and the modeling of these is discussed more thoroughly in the next section. If the goal is to consider as many architectures as possible, then any

component interface could potentially connect to any other component interface. In previous approaches, emphasis is placed on modeling the compatibility between interfaces (even of the same type). In this investigation, a slightly different approach is taken. Instead of relying on explicit structural constraints to describe interface compatibility, connections are allowed between any interface of the same type and then analysis of the system's behavior is used to deduce whether this connection is appropriate. Common connections are captured in connection templates to reduce the number of spurious connections that need to be investigated.

Now that the structure of the decision has been establish, the next section describes a language for representing that decision.

3.4 Defining a Language for Architecture Selection Decisions

In the previous section, a foundation for modeling an architecture selection decision was presented. In this section, a language is defined to capture the different elements needed to model the decision as well as the relationships between these elements. It is absolutely crucial to capture the relationship between the elements because these relationships are needed when fragments in the model are composed or reused. More discussion on the necessary relationships and how to compose models is presented in Chapter 4.

The first step to defining a language is to express its metamodel. Here, the Unified Modeling Language (UML) metamodel is used as the foundation, along with the SysML profile. When additional elements are needed, the profile mechanism is used to add additional elements and relationships. This is similar to the approach prescribed in (Weisemoller, 2007).

To support the modeling process, additional elements are needed to clearly identify each element of the decision. SysML provides Blocks to model the structure of the decision and illustrate that it contains alternatives, analyses, and an evaluation criteria. To codify this structuring, a profile is created to clearly define each of these elements. This profile is shown in Figure 3.6. Stereotypes are defined to identify the decision, the space of potential solutions, the analyses that are needed to predict the outcomes of a particular solution, and the evaluation criteria to order the outcomes. The decision problem includes multiple analyses that describe how alternatives behave. These analyses are independent of the evaluation criteria in which preferences are included to rank order alternatives. The evaluation criteria should include some objective (a value property which can be constrained using parametrics) along with a search direction. In addition, SysML requirements can be included as part of the evaluation criteria to filter alternatives. These stereotypes can be applied to modeling elements to highlight that they are part of a particular architecture selection decision and also represent the relationships between these different elements.

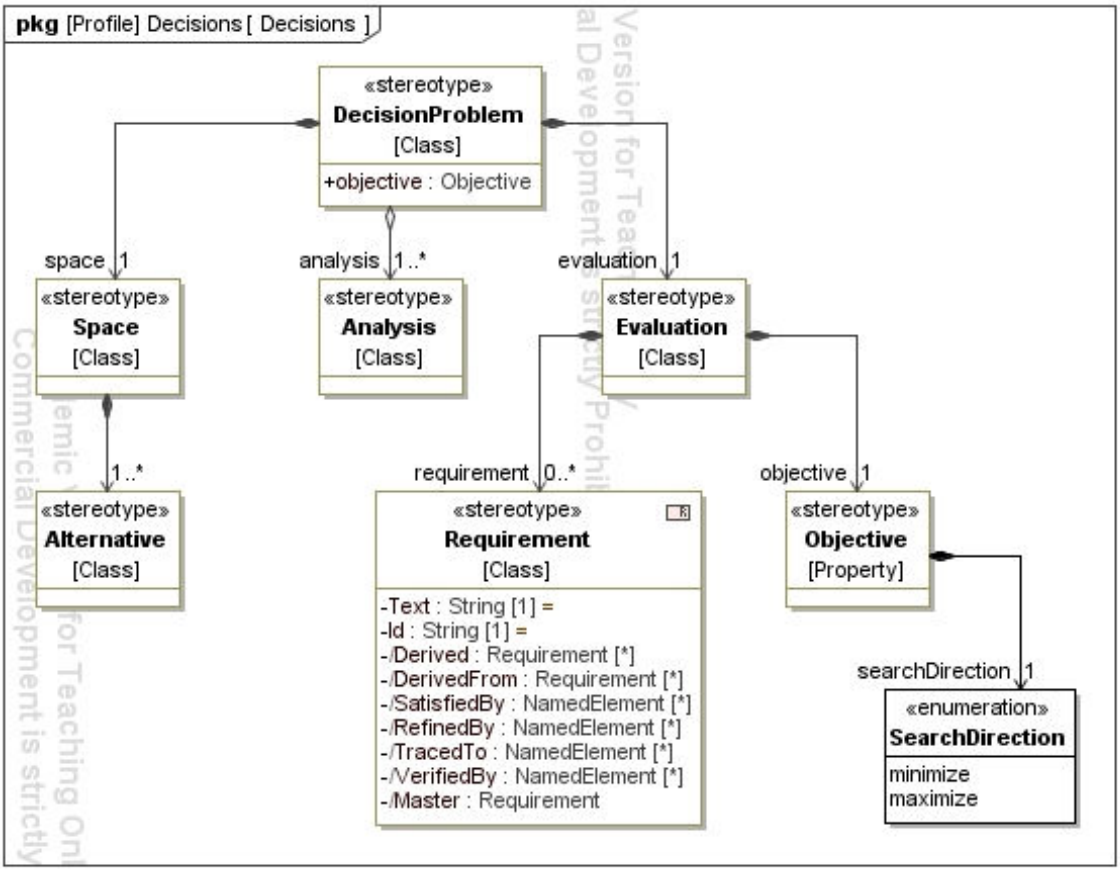

Figure 3.6: SysML Profile that defines the additional stereotypes needed to represent an architecture selection decision.

To model potential system alternatives, SysML provides a number of constructs such as the Block and Block Definition Diagram (BDD) for modeling system structure. The problem with the current practice is that the system structure modeled has been specified and is largely fixed while in this case the structure of the architecture is variable and largely unknown. It is possible that some of the architecture has been fixed, and only a part is being considered; this can occur when the design is broken down into a sequence of decisions as described earlier.

Existing SysML constructs can be utilized to represent the space of architecture alternatives in a form that is consistent with the representation in Figure 3.5, although this is a significant departure from current practice. When defining the alternative space, instead of completely specifying the structure, the approach taken here is to use the *isAbstract* property to identify elements that are not fully specified and to use multiplicities to capture variability in the number of included components just as in Section 3.3. Using this approach, an extensive space of potential configurations can be constructed and the potential structural components that appear in these configurations are clearly identified. Current SysML constructs are insufficient to define the potential connections that appear in the configurations. For this, two additional constructs are added, an «*OptionalConnector*» stereotype to describe that certain connectors are not always included in an alternative but are merely optional, and the «*ConnectionTemplate*» stereotype to group together commonly occurring optional connectors so that designers do not need to define them individually. The definition of these constructs is shown in Figure 3.7, along with stereotypes that can be used by a transformation process when flattening the modular representation into a single or set of candidate architectures. The «*FlattenedComponent*» and «*FlattenedValueProperty*» are used to specify the relationships between the original representation and any flattened representations. The use of these stereotypes along with the «*OptionalComponent*» stereotype is more thoroughly covered along with the transformation implementation discussed in Chapter 6.

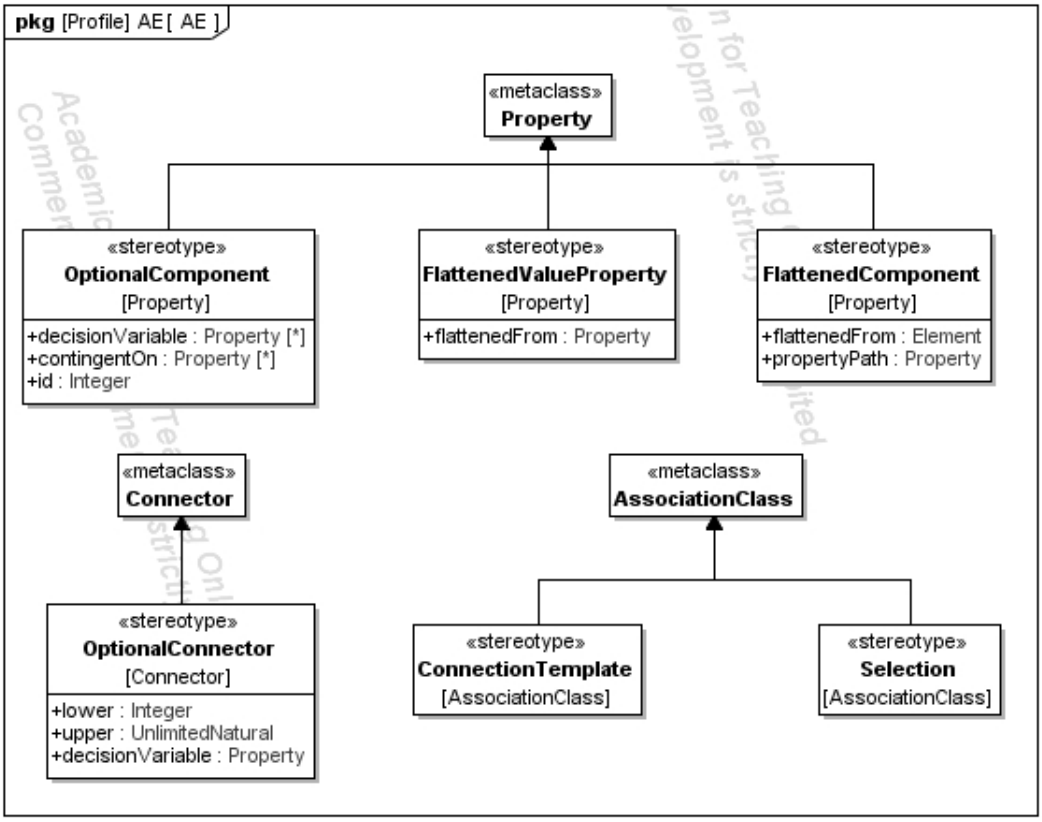

Figure 3.7: Profile for additional constructs added for defining the architecture selection decision.

As for modeling the analyses, there has been significant prior work on modeling analyses in SysML, including dynamic simulations (Paredis, 2010, Qamar, 2009), algebraic models (Peak, 2007), and various federated analyses (Min, 2011). A common approach is to use existing modeling features to capture the structure of the analysis and then define a profile to clearly distinguish these elements from the rest of the model. Often SysML parametrics are used when modeling the structure of the analyses. Some behavioral features of the analysis, for instance a particular execution sequence, can be

encoded using a wide-variety of SysML constructs, such as activities, state-machines, and so forth. More on the modeling of the analyses is described in Chapter 4.

A similar approach can be taken to model the evaluation criteria. SysML provides a number of constructs for modeling textual requirements and relating these requirements to other elements in the model. As discussed earlier, requirements alone are not sufficient to evaluate an architecture but they can be used to constrain the design space and eliminate clearly poor designs but they are not appropriate for distinguishing between good solutions. Requirements in textual form are not sufficient because verifying them requires human input. In this language, the requirements concept is extended to include the «*TestableRequirements»* stereotype, i.e. requirements that bound a certain performance attribute of the system which can be tested. These requirements have SysML properties associated with them that relate to represent the performance attribute. In addition, to define how each performance attribute should be measured, virtual *Tests* are defined. These tests are defined to be independent of any system alternative. When defining a test, the assumption is made that all considered systems will provide the same interface to the environment. This interface is the interface referred by the tests when describing interactions with the system. The values "measured" by these tests can then be included not only in the requirements but also in the evaluation criteria. This also provides a clear definition of the performance attributes that can go into the evaluation criteria and also the analyses that should be generated to execute these tests. This is described in more detail in Section 4.4.

The tests are broken into two categories: those that measure performance attributes that relate only to the structure of the system, for example its cost or weight,

and those that measure performance attributes that relate to the systems behavior over time, for example how quickly the arm actuates. Structural tests are defined by identifying the relevant structural aspects and how they can be composed into a single metric. Tests that relate to behavior are considered state-based tests, each state places some constraints on the systems behavior, and states are combined together to describe the system's behavior through time. This is similar to the state analysis concept (Ingham, 2006). These state-based tests have two parts: a test context where the structure of the test is defined and a test process where the execution order of the test is defined. The test structure is represented using SysML structural constructs while the process is defined using activity diagrams or state machines. The profile for defining «*TestableRequirements*» and «*Tests*» is shown in Figure 3.8. A simple requirements diagram showing the relationships between these two constructs is shown in Figure 3.9. In this figure, a high-level requirement is decomposed into two testable requirements, the mass and cost of a system. These are verified by tests. The performance attributes *totalMass* and *totalCost* are actually properties of the tests. The performance attributes are owned by the test because although these values are related to the system alternative, the test contains the definition of how these properties are measured.

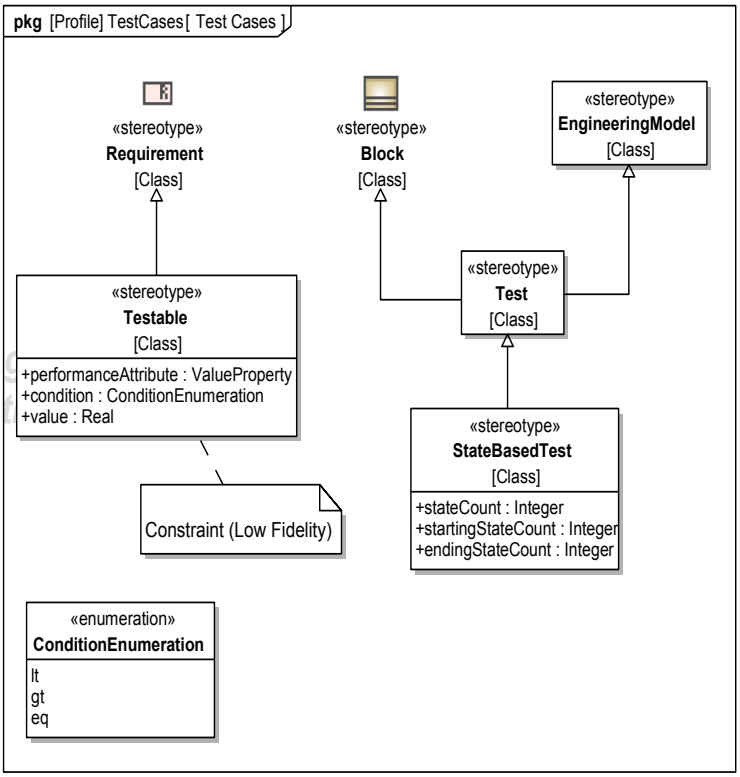

Figure 3.8: SysML profile for defining testable requirements.

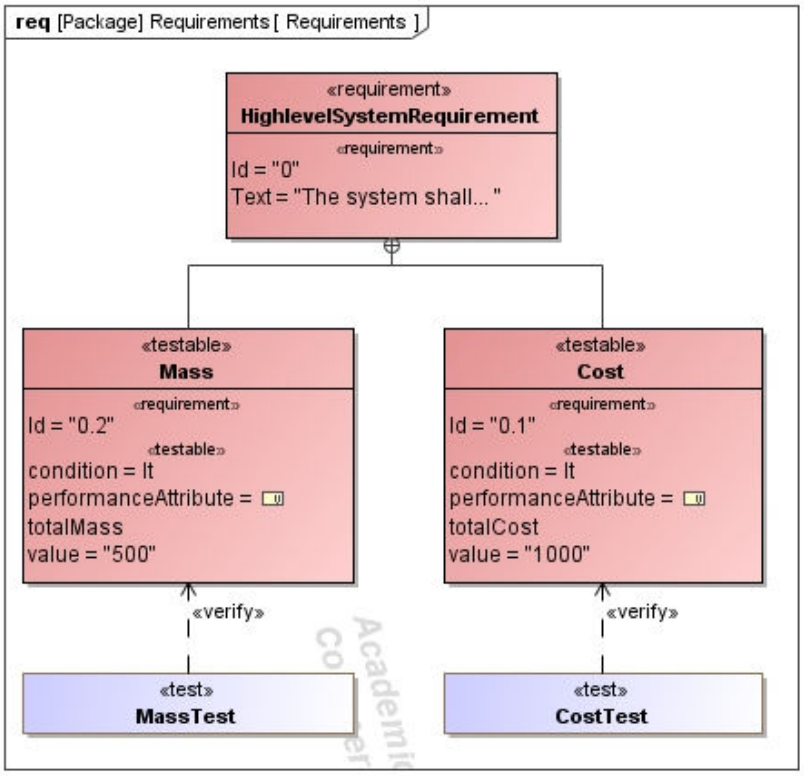

Figure 3.9: Relationship between testable requirements and test cases

To distinguish between good solutions, a value-driven design approach where an objective function is maximized (or minimized) is more appropriate than relying on requirements. To model the objective function, SysML parametrics are used. Parametrics are used to define how relevant performance criteria are transformed into a single overarching objective, whether through multi-attribute utility theory or some form of demand modeling. To address this in the language definition, the evaluation criteria can be both a set of requirements and an objective function. The requirements can be used in conjunction with some analyses to eliminate clearly poor solutions so that further computational resources need not be wasted (Moore, 2011). The next section provides more detail on the test definition.

3.4.1 Defining Tests

In order to clearly represent how a performance attribute is measured, the «Test» construct is defined. The «Test» construct defines a virtual test; the test defines the environment around the candidate architecture during the test (the inputs to the test) and which performance attributes should be measured (the outputs). The test owns the performance attributes or metrics it measures. These attributes are directly influenced by the context the test defines and are therefore owned by the test model instead of the system model. A test may relate to multiple testable requirements or the performance attributes may be included as part of a more comprehensive overall objective. Therefore, there are usually multiple tests that need to be executed for any particular architecture alternative.

One important characteristic of a test is that it is defined independently of any particular architecture. Instead, the test refers to the abstract system. The abstract system captures the interfaces any candidate architecture will have with the environment. The test consists of two parts. The first part is a definition of the test's structure that captures how the abstract system interacts with the outside world and where virtual sensors would appear. A simplified test context example is shown in Figure 3.10. The system being designed is labeled an abstract system; this is an *abstract* block with no defined internal structure. The definition does include interfaces which are common among all potential system instantiations. These interfaces are connected to test probes which measure some aspect of the system. To provide this some real world context, imagine these probes are similar to connecting a flow meter in line with a pipe to measure the flow.

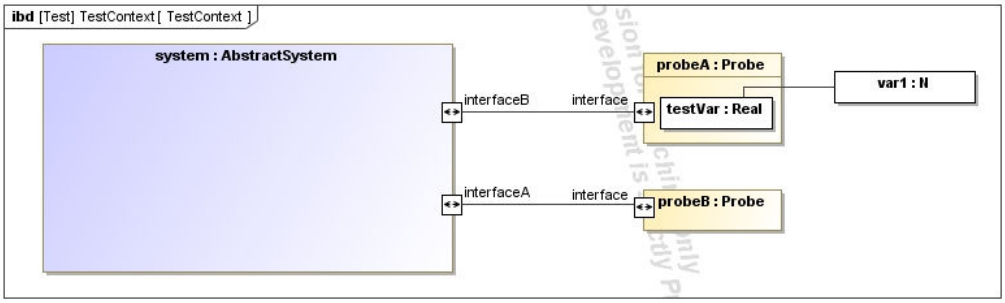

Figure 3.10: Simple test case. The system under test is connected to two testing probes (sensors).

The second part is a test procedure (often called a test protocol) which is executed to run the test. This procedure describes the dynamic behavior of the test: how the inputs to the system change over the course of the test and when to measure certain performance attributes. For the excavator, to measure the amount of fuel consumed during a certain operation, the test procedure would include the definition of the desired behavior during the operation and also when to measure the fuel in the tank (at the beginning and end of the test). Simply specifying that the attribute of interest is the fuel in the tank is not sufficient; the context is just as important.

A particular candidate architecture is modeled as specializing the abstract system; therefore the candidate alternative has the same interfaces to the environment as the abstract system and the procedure defined on the abstract system can be translated to apply to the candidate architecture because they share the same inputs and outputs.

The limitation with using the abstract system concept is that it requires all potential architectures have the same interface to the environment. This works well for the excavator example because all potential architectures must be able to suitably actuate the excavator's digging arm and always connect to the digging arm in the same locations regardless of the structure of the alternative. If an even broader space of architectures is

considered, the question becomes whether such a system boundary could be defined for that entire space. One option is to include a union of all potential interfaces as part of the abstract system and then each architecture could realize only the appropriate interfaces. The problem with this approach is that some of the interfaces would be left unrealized and tests that reference those interfaces could potentially be incorrectly specified for those architectures. Another option is to redraw the abstract system to include only common interfaces. This would require some refactoring of the boundary definition, but it seems like the more promising option.

The tests are defined in SysML using two very different formalisms. The structure of a test is defined using SysML Blocks along with the relevant constructs that usually appear on block definition or internal block diagrams. The procedure for the test is defined using UML activities. Since multiple tests may use the same structure and vary only in the process definition, the test structure can be defined in a reusable way using a test context. Then, any test using the same structure can specialize the test context and inherit this structure. This structure can also be slightly redefined as necessary using SysML's redefinition constructs. To define the test procedure, a SysML activity is created which encompasses the entire procedure. SysML actually supports multiple formalisms which would be suitable for defining the test's behavior, but the activity formalism was chosen because it seems to be the most intuitive to designers and also because constructs exist to relate the activity elements to the test context.

The SysML activity is defined to be the classifier behavior of a particular test. In the activity, the procedure of the test is defined using various actions. The activity formalism also has primitive actions that are included in SysML and derived from UML.

These primitive actions provide constructs that are used to reference features of the test structure, for instance the *readSelf* and *readStructuralFeature* constructs allow the activity to read the value of particular properties in the test structure. The *addStructuralFeatureValue* allows the activity to set the values of particular properties in the test structure at the appropriate stage during the test.

In a test procedure, there are some patterns that are used repeatedly. Again, these patterns are organized into their own activities and referenced using *callBehaviorActions*. These same patterns may also appear across multiple test procedures. In order to facilitate this reuse, some of the activities are owned by the test context and inherited by the actual test definition. An example activity for reading a particular value of the text context is shown in Figure 3.11.

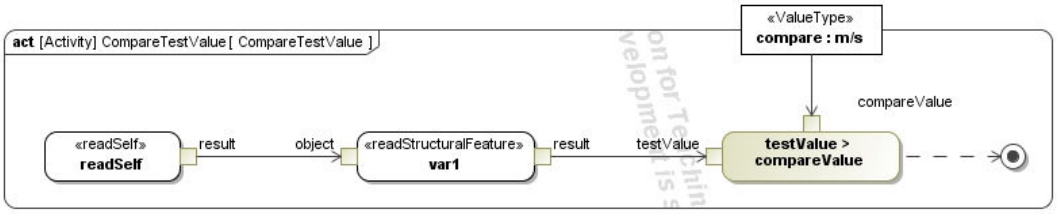

Figure 3.11: Utilizing readSelf and readStructuralFeature actions to compare the value of var1 to a test value.

The definition of the tests explicitly captures the evaluation criterion and this criterion directly affects the choice of the promising (or the best) architecture. Therefore, it is important to correctly choose the appropriate tests and specify these tests in a manner that is consistent with the designers' expectations for the system. In current systems engineering processes, test engineers design tests to verify that a particular system alternative performs as expected. The question is whether test engineers and other

domain experts can formulate tests in an architecture independent fashion. In Chapter 7, tests for a hydraulic excavator are created in a fashion that is independent from any particular hydraulic subsystem implementation by specifying the behavior of the system without constraining how this behavior is achieved. A similar approach could be used in other domains to specify what the system is expected to accomplish. The ideation of this system behavior (or use cases) is already a part of traditional systems engineering processes and the knowledge created during the process could be used to inform the creation of the tests.

3.4.2 Defining the Space of Solutions

Once the designer has created the requirements and selection criteria, the next step is to define the space of potential solutions. When defining the space of architectures, the designer needs to capture two basic facets: all of the potential components that can be used as part of the architecture and all possible connections between these components. Here, the language uses existing SysML constructs to define these. As in traditional systems engineering processes, the statement of requirements can guide the designer in defining this space of solutions and choosing the appropriate components to include. Unlike traditional processes, instead of needing to pick a particular set of components that will meet all requirements, the designer only needs to pick potential components that are applicable.

For the components, SysML offers the Block construct to define the potential component types. These potential types are captured in a component library so they can be reused for different architecture selection decision definitions; these components and the model library are discussed more thoroughly in the next chapter, Chapter 4. There are two distinct parts to the definition of a space of solutions:

1. The potential components that can be included in the solution. These are represented by using aggregation Associations from the abstract system to particular components. The multiplicity of the component side of the Association describes the number of potential components that can be included. In addition, if the included component is an abstract type, the implicit assumption is that any concrete type specializing that component could also be included in the system, although for each usage of an abstract component only one concrete component can appear in the final system specification.
2. The potential connections between components that can appear in the solution. In this representation, these are captured in connection templates so that designers do not need to specific each potential connection separately. These connection templates take the form of AssocationClasses between components.

As an example, consider Figure 3.12 where an abstract system is defined to include some number of abstract components. This defines a very broad design space, namely that the abstract system can include almost any potential components. This is defined by the aggregation Association between *AbstractSystem* and *Component.*

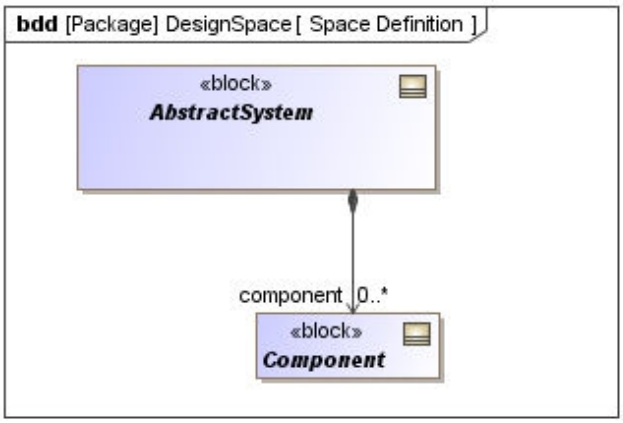

Figure 3.12: Simplified definition of a potential system

To contrast this, consider the more refined design space definition in Figure 3.13. Here, instead of considering the whole system the problem has been scoped to only include a subsystem of interest with the other subsystems being fixed. This is represented by the other subsystems not having the *isAbstract* property (in SysML syntax, this means the names are not italicized). While the abstract system may still include any number of components, it must now also include exactly one frame component. Using these mechanisms, the potential components included in the system can be scoped to be as broad or refined as desired by the designer.

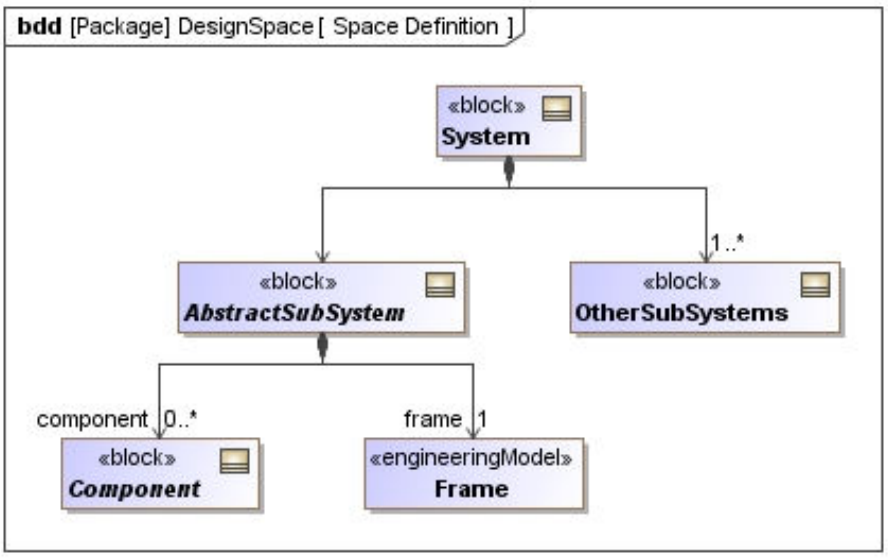

Figure 3.13: Simplified definition of a potential subsystem

In some cases, it is also desirable to group multiple components into more complex functional units; these function units reflect common combinations of components. By gathering together common components, the designer can facilitate reuse at a higher-level of abstraction while also simplifying the resulting exploration problem because the solver does not need to rediscover these common configurations each time. In the hydraulics domain, pumps are often connected to hydraulic tanks which store the hydraulic fluid. By combining these into a single "Power" functional unit, the designer does not need to include both the pump and tank in the configuration each time. In addition, the common configuration between the pump and tank can be applied in each potential architecture configuration; during the solution process this configuration does not need to be rediscovered. More discussion on combining multiple components into functional units and subsystems is discussed in Chapter 4.

To define common connections between different component types, AssocationClasses are used. Within the AssociationClass, fine-grained relationships

between the individual interfaces of the components can be defined. These AssociationClasses can appear between both abstract and concrete component types. These are stereotyped with the «*ConnectionTemplate*» stereotype to allow them to be easily identified. An example is shown in Figure 3.14.

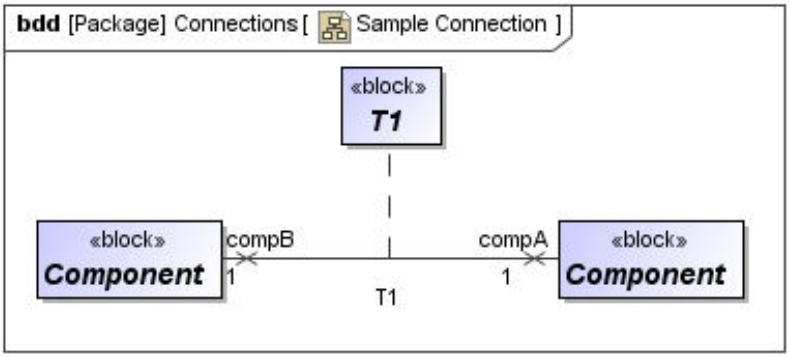

Figure 3.14: Simplified connection template between two components

As will be more thoroughly discussed in Section 4.2.1, components can be defined at various abstraction levels. For instance, a generic pump component type may be specialized into a fixed-displacement or a variable-displacement pump component type based on functionality, or a gear pump or vain pump based on implementation. The same is true for the definition of the connection templates. For example, one connection template may capture that any pump is always connected to a tank. Another might capture that a load-sensing variable-displacement pump can only be connected to a certain type of load-sensing directional control valve. Allowing connections at different abstraction levels and allowing more specific components to inherit connection templates related to more abstract components reduces the difficulty for designers in encoding potential combinations. In this example, the potential connection from pump to tank would only need to be defined once, not for every specific pump type.

Although not explicitly considered in this investigation, there is also the opportunity to explicitly capture constraints between different components and connections as part of the space of solutions. These constraints could be represented as first-order logic which is more thoroughly discussed in Section 5.2.2.

3.4.3 Capturing Domain Knowledge

Once the evaluation criteria and space of potential solutions are defined, the next step is to capture the designer's knowledge about how a potential solution performs so that it can be evaluated relative to the provided criteria. In traditional SE processes, domain experts would manually create the necessary analyses in order to evaluate a particular solution (Sage, 2000a). Since there is a potentially huge space of solutions, having a designer manually create the necessary analyses would be extremely time consuming. Instead, the approach taken here is based on modularity and composition: the expert represents her knowledge about the domain in component-level models. These component-level analysis models are related to the structural definitions of the components. When the structural components are composed into a particular architecture, the component-level analysis models can also be composed.

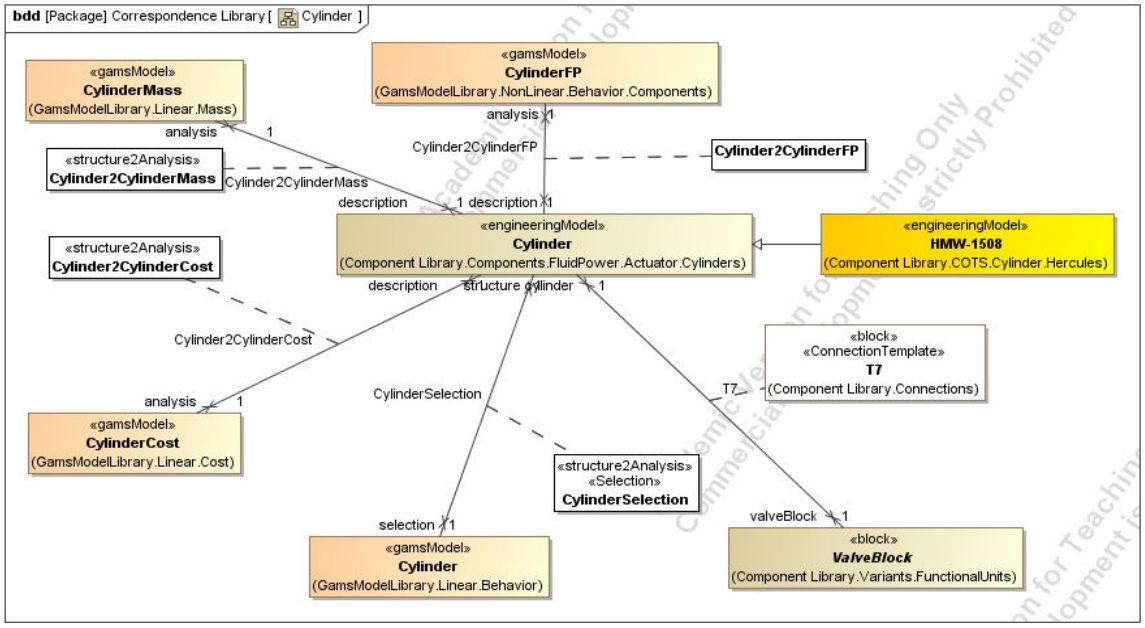

Figure 3.15: An amalgamation of models related to the structural cylinder. These different relationships are represented using *AssociationBlocks*.

The component-level analysis models are defined within SysML using SysML blocks. It is important to capture the analyses within SysML because this allows the analyses to be related to the structural definitions in a single common model. More discussion on the topic of using SysML as a common model can be found in work by Shah *et al.* (Shah, 2010c). In some cases, these blocks actually refer to analyses external to SysML. Since SysML was designed to be used as documentation, it cannot be used in a stand-alone way to perform analyses. Therefore, additional profiles have been defined for SysML to allow the representation of analysis models that can be transformed into representations that are then solved in external tools. One such example of this is the SysML-Modelica specification where Modelica models can be represented in SysML and then exported to a Modelica solver (Paredis, 2010).

To link the structural component models to the component-level analysis models, SysML *AssociationBlocks* are used. These blocks also contain connectors that link the parameters and interfaces of the analysis model with the corresponding parameters and interfaces of the structural model. For each type of structural model, there may be multiple relevant analysis models that are linked that capture different aspects of the components behavior. For instance, a number of properties for a hydraulic cylinder can be analyzed, the cost, mass, dynamic performance, or heat generated. Each of these might require different analyses which are related to the cylinder. The models related to the cylinder via AssociationBlocks are shown in Figure 3.15. In order to generate a system-level analysis model, the appropriate component-level models must be composed. In order to differentiate between the different analysis models, a classification scheme is used based on aspects (Kerzhner, 2011). The aspects are made up of orthogonal characteristics that can describe an analysis model, such as the representation syntax or analysis type. These aspects are organized into a hierarchy which can be expanded as needed by the designer. More discussion on the definition of aspects is provided in Section 4.2.3. The relationships between structural and analysis models are stereotyped with the «Structure2Analysis» stereotype. These relationships are also associated with the appropriate aspects. This facilitates finding all the appropriate *AssociationBlocks* that relate to a particular aspect when it is time to compose analysis models from the structural model. Capturing and composing domain knowledge is the focus of the next chapter, Chapter 4. Also, the domain knowledge included in the model libraries as part of this investigation can be found in Appendix A.

3.5 Discussion

The goal of this research is in part to shift the use of SysML and similar modeling languages from simply documenting the deliverables in a systems engineering process to supporting designer decision-making during the process. In order to accomplish this, designers must be able to clearly represent the decisions they are considering, the alternatives they are selecting over, knowledge about the outcomes, and their personal evaluation criteria. From this representation, model transformations can be used to generate a number of different analyses that can guide the design making during the design process as will be demonstrated in the following chapters.

The other important issue to consider is how this representation supports the rationality of designer's decision making. As described previously, decision theory provides a strong theoretical foundation for rational decision making. The difficulty is in implementing decision theory in real-world design processes. Some of the difficulty is in managing the number of available alternatives and relevant analysis knowledge, which has been the main focus of this Chapter.

For decision theory to be applicable, decisions must be made by a single decision maker. Arrow proved that any aggregation scheme that aggregates preferences (which is not a dictatorship) can lead to inconsistent results (Arrow, 1963). Therefore, multiple designers using some aggregation scheme such as voting to make decisions about a design alternative has the potential to lead to poor results. Current research into addressing this problem provides two potential solutions: a game theory-based approach (Hurwicz, 1960) or an aggregation approach. In the game theory-based approach, a single decision maker sets the rules of a "game" in which other designers make their own

decisions, but because of the structure of the game maximize the utility of the original decision maker. Here the decision is not in selecting a particular design alternative but in selecting the structure of the game. In the aggregation approach, knowledge from multiple experts is aggregated together into predictive models and then a single decision maker uses that knowledge to make a final decision.

The modeling approach presented here is needed to serve as the foundation of an aggregation-based decision approach. The explicitly modeled decision gives domain experts a starting point for capturing their knowledge and aggregating this knowledge into a single model. The best process for accomplishing this aggregation is not considered in this investigation and is left as an open question.

The other difficulty in applying decision theory is in creating the necessary predictive models. This is not the focus of this research, but it should be considered how predictive models could potentially fit into such a framework. This framework relies heavily on modularity and composition; composing together predictive models is more challenging than composing together deterministic models. One approach is to compose deterministic models as described and then use a Monte Carlo-based approach to sample these models and generate a prediction of system performance. Clearly, in early stages of the design process, employing Monte Carlo-based methods where many function evaluations are needed on a wide variety of solutions can be very computationally expensive. The other option is to formulate the models as predictive models by using the expectation operator throughout. This would require different composition rules in order to compose component-level predictive models into system-level predictive models, but

may be more tractable computationally. Either approach could be added to this framework.

3.6 Summary

In this chapter, a language for modeling architecture exploration problems as architecture selection decisions was presented. The constructs included in this language are specifically tailored to model the knowledge needed to make the decision, such as the alternatives being considered, how to analyze those alternatives, and how to evaluate the outcomes of the analyses. Modeling the problem is the first step toward the use of computational tools to support the selection of system architectures.

CHAPTER 4:

MODEL LIBRARIES AND COMPOSITION

In this chapter, a generic approach is presented for capturing reusable analysis model fragments and using them to compose analysis models. This approach is based in part on previous work by Jobe et al (Jobe, 2008, Kerzhner, 2011), where reusable model fragments were organized into multi-aspect component models (MAsCoMs). Although the previous work provided some foundation for reusing model fragments, in this chapter many of the practical issues encountered by this earlier work are addressed in an operational approach. The goal of this chapter is to address RQ2:

RQ2. How can domain-specific synthesis and analysis knowledge be captured and organized effectively to allow for composition and reuse?

The previous chapter has focused on providing a language for modeling architecture selection decisions explicitly in information models. Without addressing RQ2, utilizing this modeling framework for real-world examples is not practical. There are two major considerations:

1. Significant effort is required to model the knowledge that is included in the architecture selection decision and there is significant overhead in creating the model, but this overhead can be mitigated by reusing some of the encoded knowledge over different problems.
2. In order to analyze each potential architecture alternative, an executable simulation or optimization is needed. When searching a space of architectures, if these analyses are

not automatically generated then a designer needs to manually create an analysis for each alternative.

To address these issues, hypothesis 2 is presented:

H2: Designers could use modularity and composition along with model transformations to reuse knowledge encoded in models within and across design problems.

To support this hypothesis, a framework is presented for capturing knowledge in reusable model fragments that are organized into model libraries. This framework is then used to compose knowledge from these fragments into a system-level analysis model. The basis for this hypothesis is applying the concepts of *modularity*, *reuse,* and *composition* to shift the cost-benefit balance in favor of explicitly modeling the problem by reducing the modeling costs.

The focus is specifically on the reuse of analysis knowledge, the knowledge used to create analysis models from the structural representation of a system. Analysis models are ubiquitous in current systems engineering practice; they are used for predicting the behavior of components and systems from different viewpoints. They are interesting from a reuse perspective because they can be reused not only from one design problem to the next, but also in multiple design iterations within a single design problem. This chapter presents a framework in which analysis knowledge is systematically encoded. It then supports the composition of this analysis knowledge to generate system-level analysis models from system-level structural representations.

In this chapter, the focus is on how to capture reusable fragments in libraries, how to organize these fragments, and then how to identify the appropriate fragments and

compose them together. As a starting point, the problem is viewed for the case of only a single architecture. In Chapter 6, the concepts are extended into a transformation approach that transforms the architecture selection decision into a mathematical programming problem. This transformation approach and the illustrative examples presented in Chapter 7 provide further support of hypothesis 2.

The rest of the chapter is outlined as follows: the next section describes previous and related work related to modularity and reuse in systems engineering. Then the approach used in this investigation to capture reusable knowledge in model libraries is presented. Section 4.3 presents the implementation of this approach in SysML. Section 4.4 presents the transformation approach for composing models. Then Section 4.4.1 presents a practical example with a simple hydraulic circuit.

4.1 Prior Work in Modularity and Composition

Many have recognized that design elements are often modular and have the potential to be reused. Baldwin and Clark (Baldwin, 1999) consider the use of a design structure matrix, task structure matrix, and modular operators to capture design modularity. Eppinger *et al.* (Eppinger, 2000) also identify that many systems can be decomposed into modules, although they note that some systems are integrative in nature and cannot be decomposed. Integrative systems avoid the overhead of modular interfaces which can improve performance (Ulrich, 1991) but may be more difficult to maintain and also are less likely to have reusable elements. Gershenson *et al.* (Gershenson, 1999) consider modularity as it applies to the entire life-cycle of a product design. They claim that all components that are of the same form (based on function and interface) will undergo the same life-cycle processes. The abstract level of the component being

considered has an effect on the commonality between life-cycle processes. This also holds true for the selection of a modular analysis model to predict the behavior of a structural component.

There has also been a shift toward analysis modeling approaches which are modular in nature. Usually these approaches allow designers to develop their models in a hierarchical fashion, constructing more complex models by combining and connecting simpler models at their interfaces. This can be seen in the multi-domain dynamic simulation area with declarative, object-oriented modeling languages such as Modelica (Modelica Association, 2005). Similarly, in the discrete-event simulation area, models are connected via well-defined inputs and outputs using formalisms such as the Discrete Event System Specification (DEVS) (Zeigler, 1999) or tools such as ARENA (Kelton, 2002).

The idea of reusing design knowledge by storing the knowledge in a repository has also been proposed in the past. The NIST Design Repository (Szykman, 1998) was one of the first efforts in this area. Further development of the knowledge representation underlying the NIST Repository resulted in the Core Product Model (CPM) (Fenves, 2008). The CPM is a high-level meta-model in which the core elements for representing products in design (i.e., form, function, and behavior) are identified and related to each other. The goal of the CPM is to provide a common foundation for product representation that can then be further refined as needed, e.g., for engineering analysis (Bajaj, 2007a, b), for manufacturing process planning, for functional decomposition (Kopena, 2003, Stone, 2000), or for assembly planning (Rachuri, 2005). The models developed in this chapter are informed by the concepts of the CPM, although the focus is

on more specific constructs for *system behavior*. Here, behavior is to be interpreted as any type of characteristic that can be predicted based on the form, distinguishable by many behavioral aspects, including function.

The goal of the CPM and the information modeling part of this investigation can be loosely described as defining an ontology for design (or more specifically in this investigation systems design), although the design domain is very broad and it is unlikely that any single ontology will sufficiently capture it. An ontology is a formal data model for the concepts and the relationships between these concepts in a certain domain of discourse — the domain of *design* in this case. Most of the research in this area shares the perspective that at the foundation, one should distinguish between form, function and behavior. Examples include the work by Umeda *et al.* (Umeda, 1990), Kitamura and Mizoguchi (Sasajima, 1995), and Horváth *et al.* (Horváth, 1998). However, *system behavior* has been the focus of investigation in only a few previous publications.

The most extensive previous research on characterizing behavior in engineering analyses was performed by Grosse and coauthors (Grosse, 2005). They organize the knowledge about engineering analyses models into an ontology, which includes both meta-data (e.g., author, documentation, etc.) and meta-knowledge, such as model idealizations and the corresponding justifications. A similar, although less extensive, meta-model for engineering analysis models has been developed by Mocko *et al.* (Mocko, 2004).

In this section, this past work is expanded to enable reuse of engineering analyses in the context of large systems engineering efforts. In this respect, two extensions are important: First, the engineering analyses need to be related to the form (e.g., component

geometry or system architecture) at a fine-grained level (Peak, 1998). Second, the analysis models for components and subsystems must be formulated in a fashion that allows for composition so that a large number of different system architectures can be explored quickly (Paredis, 2001).

Relating analysis models to structural form has been addressed in work on Design-Analysis Integration (DAI) (Peak, 1998), although these relationships are not captured in a form that is conducive to automated composition of analysis models. Peak *et al.* relate the parameters of analysis models to parameters of design models when using Constraint Objects (COBs) or, more recently, using SysML parametric diagrams (Peak, 2007). In this investigation, the relationship between structural models and analysis models is captured at the level of individual components (see section on Fine-Grained Design-Analysis Relationships). These relationships are maintained when the components are composed into larger systems, providing a template for reuse. To enable composition, additional knowledge is needed both about the model interfaces and about the composition process. Wallace *et al.* (Wallace, 1998) also consider composable models. They note that a modular, composable analysis approach allows multi-disciplinary problems to be broken down into modules that can be assigned to specialized teams.

4.2 Capturing reusable Analysis Knowledge in a Model Library

A model library contains useful model fragments and information which can be composed into more complex models. In this case, the model library contains knowledge at a component-level about analysis models. The multi-aspect component model

(MAsCoM) is used as the basis for the specification and organization of this model library.

Several key pieces of knowledge are captured in this model library:

1. An enumeration of the available analysis models.
2. A mapping between the available analysis models and the corresponding structural models.
3. How the parameters and interfaces of the analysis models related to the parameters and interfaces for the structural components.
4. Which analysis models can be connected together and how they should be connected together via their interfaces.

The organization of this library takes into account the general view of systems engineering problems used throughout this investigation, that systems contain components that are connected together via their interfaces. Analysis models are organized by component type because it follows naturally from the definition of an architecture selection decision and also allows designers to conveniently view and review the library. Whenever a particular component is chosen, a designer will immediately be able to identify all the analysis models that have been previously used to analyze that component or describe its behavior in a larger system. The components themselves are organized in a taxonomy so that the user can easily browse from general classes down to very specific instances of components. At each level, the component model can be linked to all the relevant engineering analysis models.

However, the number of such models could be very large, so that an additional method of organization is desirable. To facilitate the task of selecting and composing

analysis models further, the analysis models are characterized based on one or more *aspects*. In Aspect-Oriented Software Development (Tzilla, 2001) modularity is achieved by implementing cross-cutting concerns separately so that they can be woven into a variety of different software classes. In this context, rather than weaving models together, what is important is to identify which models are compatible with each other so that they can be composed into system-level models.

In this composition approach, for models to be compatible it is necessary that they characterize the components in a system from a similar perspective, in a compatible mathematical formalism and in the same executable language. By using a formal taxonomy of aspects, the semantics of the individual analysis models are characterized in a computer interpretable and searchable fashion.

In the remainder of this section, the details are provided for how analysis models are organized into model libraries. In addition to discussing taxonomies of components and aspects, it is explained how analysis models are tightly linked to structural components at a very fine-grained level.

4.2.1 A Library of Components

To enable the composition and reuse of analysis models, the first step is to identify and store common component and subsystem models in a model library. In this framework, these individual components and subsystems are organized into a taxonomy starting with the most abstract definitions and progressing to particular types of components and finally to particular component instances (for example, particular off-the-shelf components from a manufacturer).

The organization of the components into a hierarchy simplifies the definition of the architecture selection decision. In the definition of the decision, the designer can include components at any level of abstraction. The implication is that the design space would include the concrete instantiations of the (potentially abstract) components.

Organizing the components into a taxonomy also supports traditional systems engineering processes where components or subsystems are selected and defined in an iterative fashion. After a functional architecture is defined in classic systems engineering process, functions are assigned to components in a physical architecture (Sage, 2000b) (or, equivalently working principles and working structures are identified (Pahl, 2007)). The focus is initially on the selection of broad classes of components that share the same functionality. For instance, to implement the function of converting electrical to mechanical energy, the broad class of motors could be identified. In subsequent iterations, this broad class of components is gradually refined until a particular component is identified. At each step along the way, analysis models at different levels of abstraction are used. As the definition of the components still under consideration becomes more and more detailed, the corresponding analysis models also need to become more detailed such that the selection can continue to be narrowed down further. For instance, since an axial piston pump is a type of displacement pump, the models for the general class of displacement pumps (the parent) also apply to axial piston pumps (the child). However, constructing more detailed models of the children should be possible because more detailed knowledge is available about their structure, size, or other design properties.

4.2.2 A Library of Analyses

To enable the composition and reuse of analysis models, it is also necessary to also capture them in a model library in a reusable form. There is the potential to include analysis models in this library at many different structural levels; for instance, analysis models of an entire system could be stored in the library and then reused when the same system is analyzed in future design problems. At the other end of this spectrum, the analysis models representing fundamental behavior could be included and reused when modeling the behavior of a particular component. Representing such low-level model fragments would certainly increase the opportunity for reuse, but the composition process between these low-level fragments would be more complex. In this investigation, the focus is on component-level analysis models, those which model the components of a system. Any time the same component appears in a system, there is the opportunity for reuse. This matches with the current definition of the architecture selection decision wherein the architectures are composed of different subsets of the same set of components. Again, these analysis models are organized in a hierarchical taxonomy similar to the component models. Defining them hierarchically simplifies the definition process because less abstract analysis models can inherit many of the same properties and equations (or constraints). Also, it simplifies the establishment of connections between the structural and analysis libraries because less abstract components can inherit some of the relationships.

4.2.3 A Library of Aspects

When attempting to reuse the models related to a particular component, one needs to recognize the appropriate analysis model. To help support this process, models are

characterized using aspects. Since there are a large number of potential aspects, it is helpful to organize them in a taxonomy. The taxonomy also emphasizes that the aspects represent independent directions along which a model can be characterized. As a result, a model is typically characterized by multiple aspects simultaneously. For example, a pump model could be characterized simultaneously by the fact it models dynamic behavior, has hydraulic interfaces, and is also represented by the Modelica representation syntax.

These aspects characterize the model and thus succinctly provide the basic information needed to select an appropriate model. Additional information about the model can be defined as meta-data that is less structured, such as model documentation, development history, or prior usage scenarios. In addition, when composing multiple component models into a system-level model, the aspects provide necessary information to determine compatibility between models. For instance, to be composed, models need to be expressed in compatible mathematical formalisms and levels of discretization—it is usually not meaningful to combine a steady-state behavior model with a partial differential equation behavior model. Models that are composed also need to share compatible engineering disciplines. One set of models may describe the hydraulic behavior of a system while another may describe its mechanical structure. Having formal representations of these different aspects available is particularly important when automating the composition process.

4.2.4 Fine-Grained Design-Analysis Relationships

When attempting to compose a system-level analysis model, it is important to consider what knowledge is needed in addition to a structural view of the system. Even

though in a variety of engineering disciplines it is common to describe systems as compositions of components in a schematic diagram, the question is: what additional knowledge is needed to automatically instantiate and configure the corresponding system-level analysis models?

It is not sufficient to simply have a library of analysis models, even labeled with appropriate aspects and linked to the appropriate structure components. This only allows the identification of the appropriate analysis models; it does not allow them to be composed. In a schematic diagram, components are usually connected via their interfaces (ports), so without understanding how these interfaces relate to the interfaces of the analysis model, it is impossible to connect the analysis models in similar fashion. The same is true for properties (or variables) in the structural description. Without understanding how they relate to the properties of the analysis model, it is impossible for the analysis model to contain the same values as the structural model.

In order to support the representation of this knowledge, two additional mapping definitions are included in the modeling approach: *parameter maps* and *interface maps*. These capture additional knowledge about the relationship between the interfaces and parameters of the structural models and analysis models.

Parameter maps bind the parameter values of analysis models to the related parameters of the corresponding structural model. In the context of systems engineering, the values for the parameters need to be related to the properties of the system alternative that is currently being analyzed. Since we have associated the analysis models with components in the component taxonomy, it becomes possible to establish these

relationships also in a reusable fashion. How this is accomplished using SysML parametric diagrams is explained in Section 4.3.3.

Interface maps support the configuration of analysis models for individual components into system-level analysis models. Similar to the composition of structural models into a system schematic, analysis models can be configured into networks through well-defined port-based interfaces (Paredis, 2001), as is implemented in tools such as Simulink™ (Simulink (The Mathworks), 2008), and in languages such as Modelica. Recently, the ability to compose analysis models has even become feasible for finite element models (Bajaj, 2007a, Simmetrix Inc., 2006). As mentioned earlier, this is also the case for discrete-event models. In order to configure the analysis models, one needs to define how the ports of the analysis models relate to the ports in the structure models. This is accomplished through interface maps as is further explained in the next section.

4.3 Implementation in SysML

In this section, we present how this framework (including model libraries and relationships between model libraries) is implemented in SysML. Components are organized into a component taxonomy described in Section 4.3.1. The classification of analysis models using aspects is covered in Section 4.3.2. How the descriptive component models are related to the analysis models is shown in Section 4.3.3.

Both the structural and analysis models are represented in SysML. This allows SysML to act as a common language in which correspondences between the models can be explicitly defined. In addition, when composing new analysis models, correspondences can be created to allow traceability to the original structural model.

Using SysML as the common language does have the disadvantage of requiring an additional mapping from SysML into a language that can be interpreted by a particular simulation tool, but there are a growing number of such mappings emerging to support tool interoperability, such as between SysML and Modelica (Johnson, 2008, Paredis, 2008), SysML and eM-Plant (Huang, 2007), and SysML and the General Algebraic Modeling System (GAMS), which is similar to AIMMS (Shah, 2010b). The mapping between SysML and Modelica has recently been adopted as an official OMG specification: the SysML-Modelica Transformation specification (Object Management Group, 2010).

In order to represent the aspects for characterizing models and the relationships between structural and analysis models, some additional concepts not available in SysML are needed. Just as in Chapter 3, the profile mechanism is used to add these new constructs through the addition of several stereotypes to the model. Since SysML is defined specifically to support systems engineering, it includes modeling constructs that directly support the definition of physical architectures and engineering analyses so additional constructs are not needed for these elements. The created profile is shown in Figure 4.1. There are three new concepts added: the «*EngineeringModel*» stereotype to allow both structural and analysis models to be associated with aspects; the «*Structure2Analysis*» stereotype for identifying the links between structural component-level models and the corresponding analysis models; and the «*Aspect*» stereotype for marking aspects. This profile will be used through the following sections.

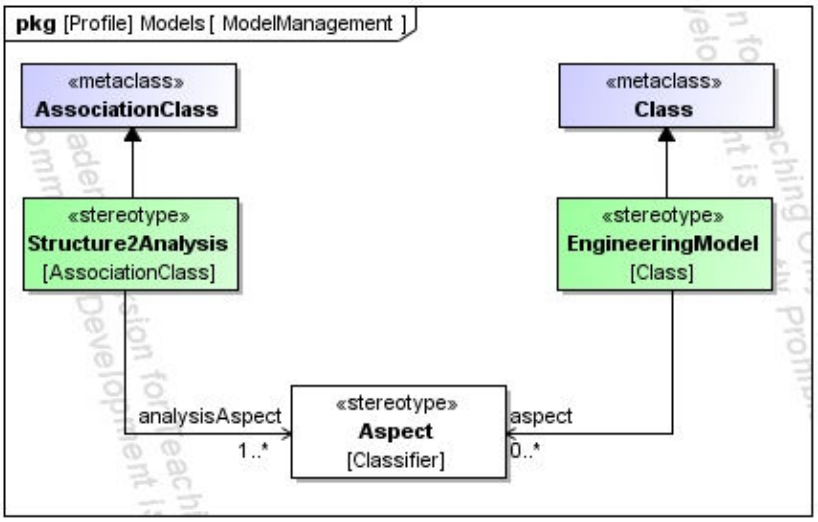

Figure 4.1: Profile for capturing correspondences between structure and analysis models.

4.3.1 Component taxonomy

The component taxonomy is represented in SysML using packages and blocks with generalization relationships to represent inheritance. Flow ports owned by the blocks are used to describe the interfaces of the components and value properties are used to describe properties of the components that can be assigned a value.

In addition to the component-level taxonomy, the component-level models can then be composed into more complex sub-systems which are organized into a different taxonomy. This simplifies the definition of the selection decision because a designer can then include a particular sub-system in the solution space instead of including all the constituting components. In addition, this simplifies the search process because the solver does not need to consider these components and connections separately.

A small section of the component taxonomy is shown in Figure 4.2; this taxonomy contains only component-level structural models. The between component models and sub-system models is a choice made by the modeler.

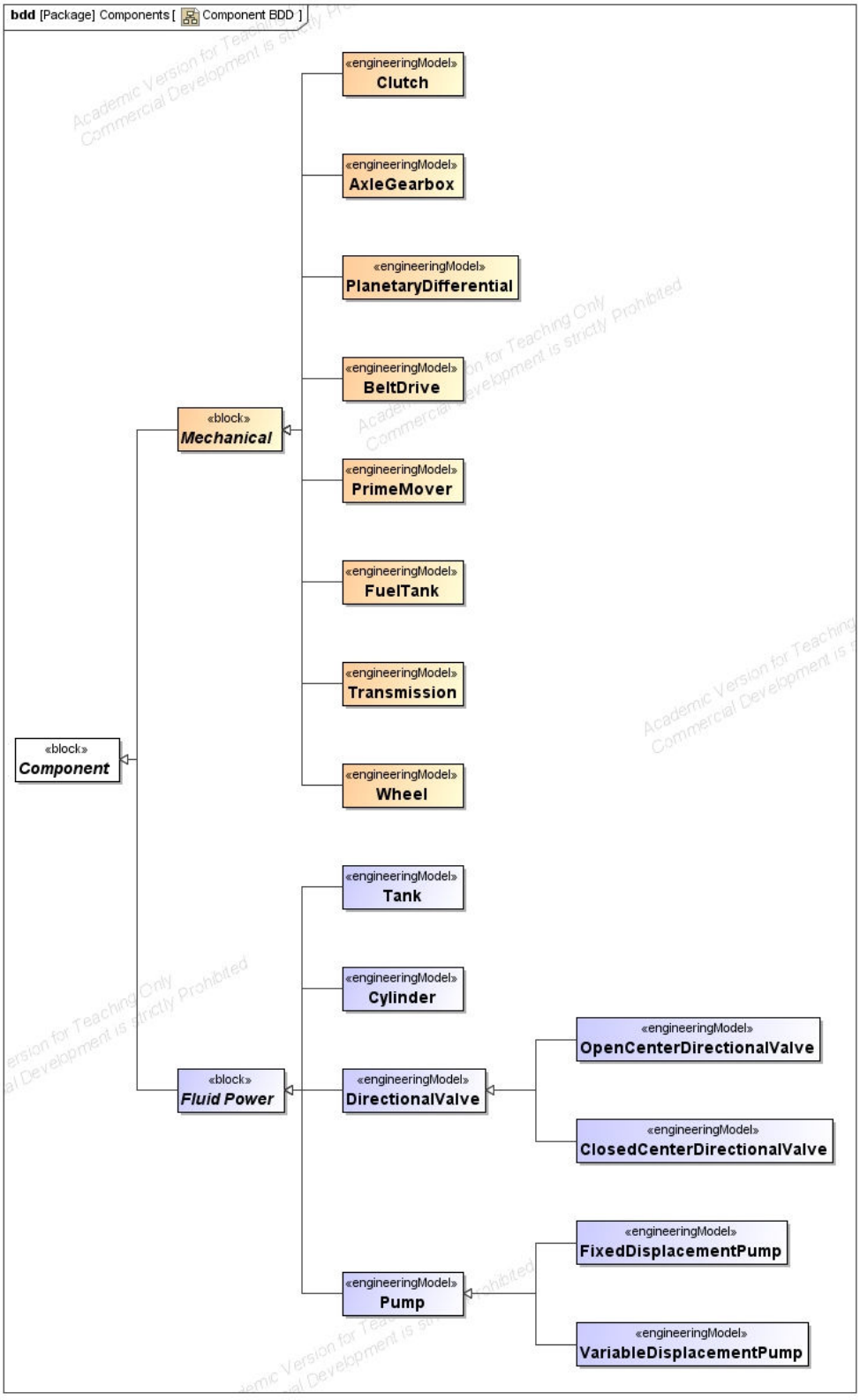

Figure 4.2: A partial view of the Component hierarchy.

As mentioned earlier, the components are organized into a taxonomy making them easier to specify and also easier to include entire classes of components in the selection decision. For instance, there are many pumps that can specialize the more general "*pump*" concept and those pumps inherit the same attributes and ports. All pumps are modeled as having a mechanical input, a housing to mount the pump, and then a minimum of two fluid ports (load sensing pumps often have three). An example of such a component breakdown is shown in Figure 4.3. The general pump is specialized into a specific type of pump (fixed displacement) and then into a specific vendor pump.

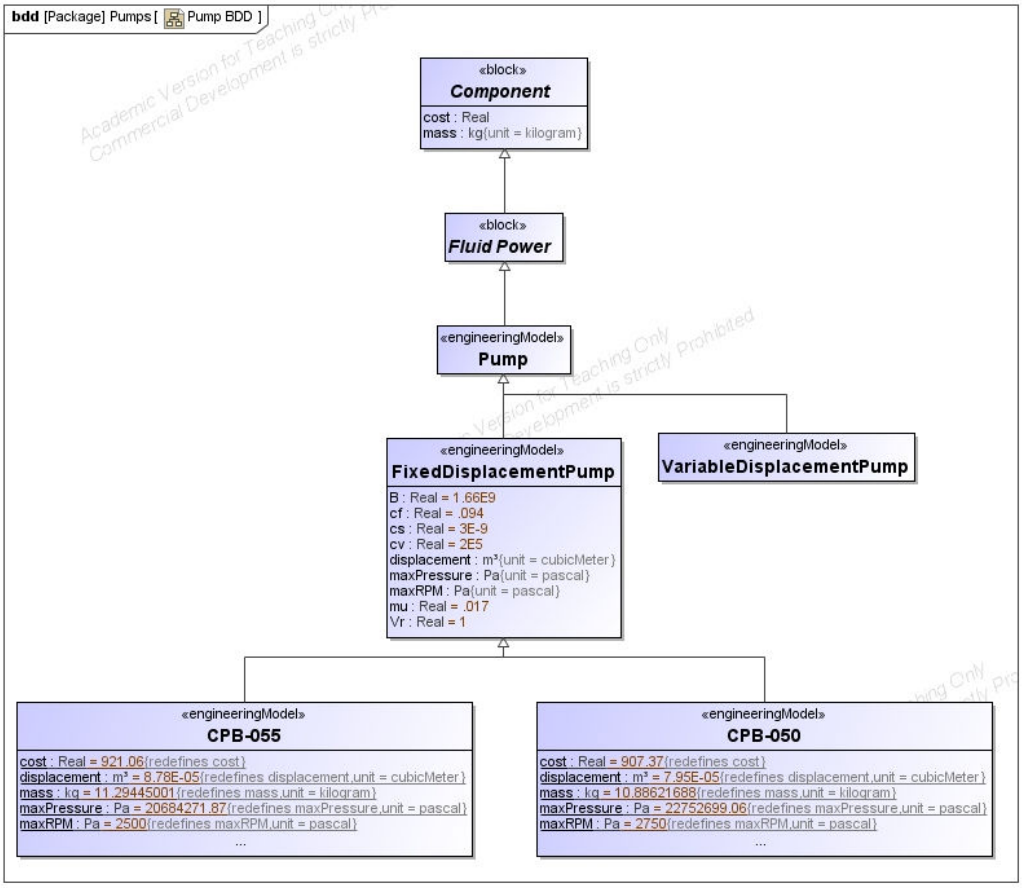

Figure 4.3: Component hierarchy of pumps

Components from this taxonomy are then composed into the system-level structural models that are then transformed into system-level analysis models.

In addition, components that are commonly used together can be grouped together into subsystems which are then also stored in the model library in the same fashion. The taxonomy for the subsystems is shown in Figure 4.4. In this case, considering these combinations of components as subsystems seems odd because they contain only a handful of components and are incapable of operating separately, instead they are labeled functional units. Figure 4.5 shows the relationship between the two taxonomies. A similar hierarchy exists in both taxonomies, with more concrete components being included in the more concrete functional units. These functional units can also be composed into the structural models.

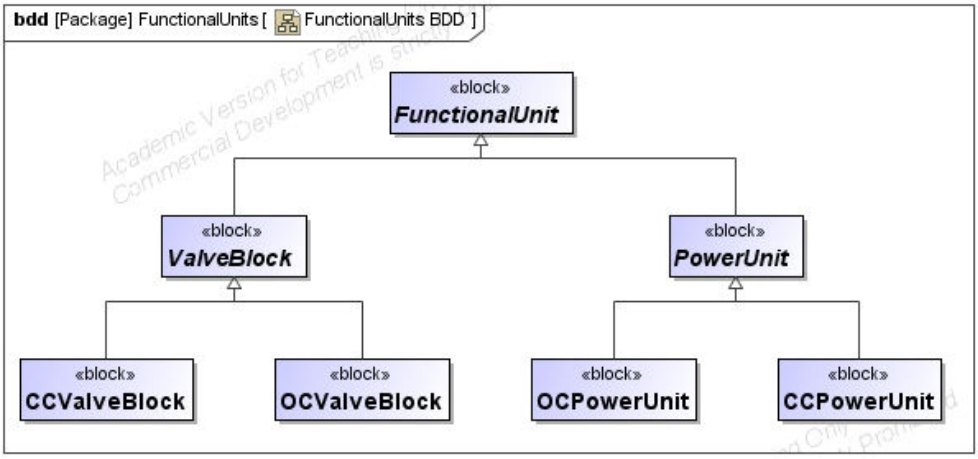

Figure 4.4: Sub-system taxonomy, sometimes referred to as functional units.

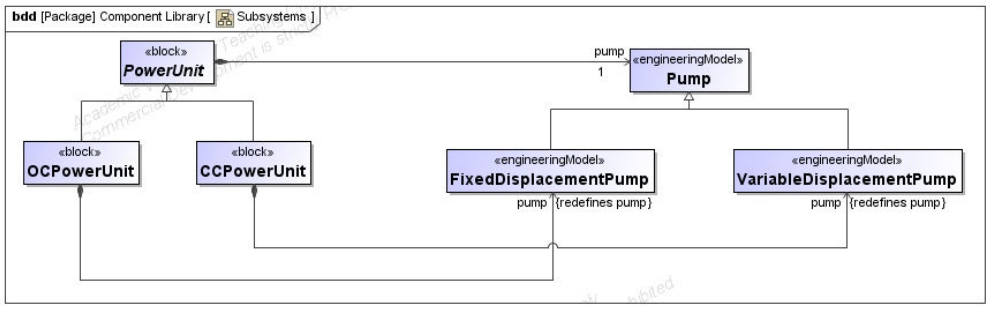

Figure 4.5: Relation between the subsystem taxonomy and the component taxonomy

4.3.2 Aspect taxonomy

The aspects are organized using a similar approach as the component taxonomy. The aspects are defined using SysML blocks. Specializations are used to order aspects from most abstract to least abstract. Each aspect is stereotyped using the *«Aspect»* stereotype from the previously defined profile. This simplifies identifying aspects during the transformation process. There are many aspects that could be considered; only a small portion of the aspect taxonomy is highlighted in Figure 4.6. As an example, the aspects categories provided describe the representation syntax of an analysis model or the type of system behavior the analysis model is capturing.

When analysis models are composed, they need to be chosen so as to have the appropriate aspects. For instance, two analysis models need the same representation syntax to be composed. Further investigation is needed to consider exactly which aspects need to match in general.

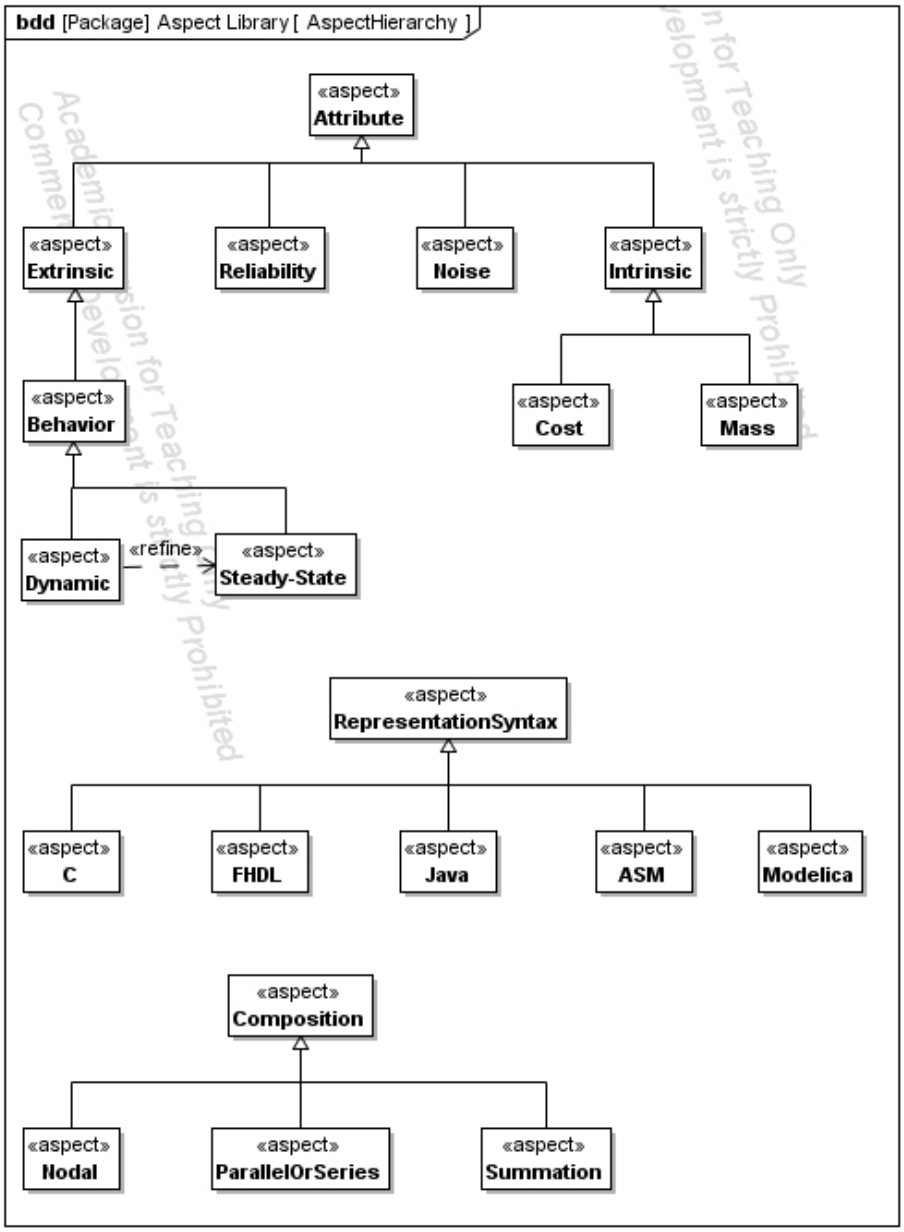

Figure 4.6: Package structure for aspect taxonomy.

4.3.3 Library of Analysis Models

In addition to capturing potential components in a model library, analysis models for these components are also needed. These analysis models can again be defined in a hierarchical fashion where less abstract models inherit many of the values and equations

of more abstract models. In this investigation, two different kinds of analyses are considered. The first are analyses directly represented in SysML, included in this set are the algebraic component models used in the mathematical programming formulation described in the next chapter. A high-level overview of the behavioral algebraic models is shown in Figure 4.7. How these models are defined is more extensively covered in the next chapter and a full listing of these models and included constraints can be found in Appendix A.

The second types of analyses are only referenced from SysML. These include any Modelica models mentioned in this investigation. A more complete discussion of these analyses is found in Section 4.5

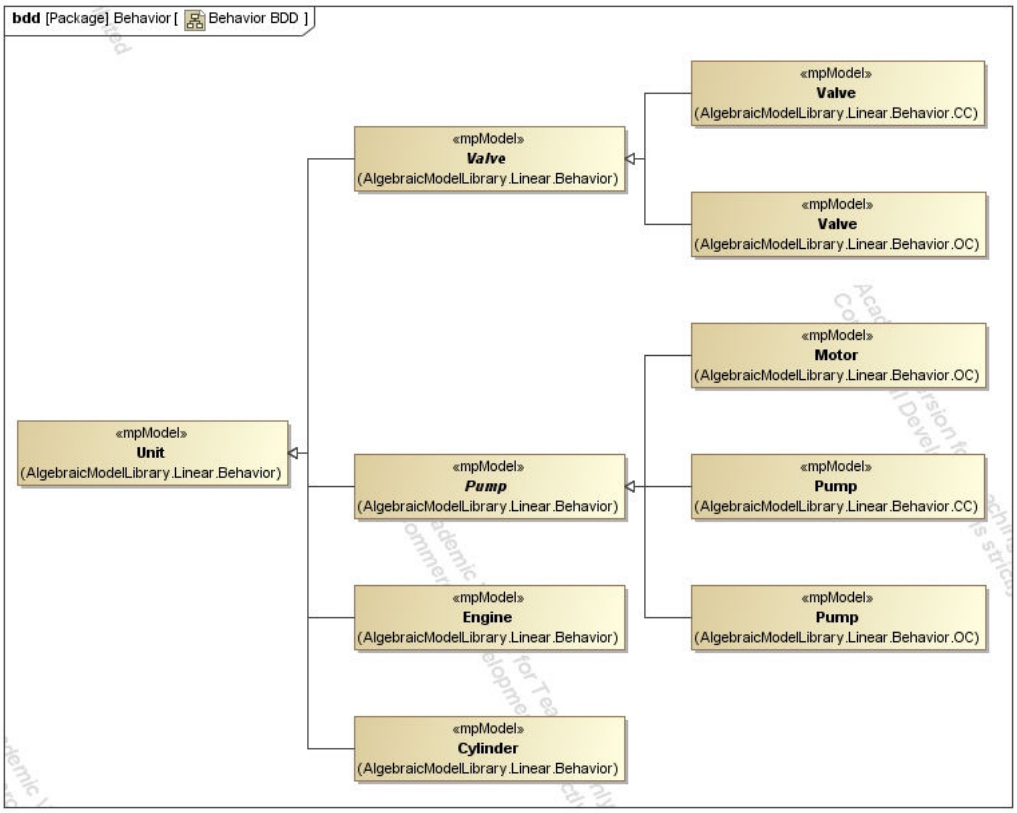

Figure 4.7: Overview of algebraic behavior analysis models.

4.3.4 Establishing Fine-Grain Mappings

SysML AssociationClasses are used to link an analysis model to a structural component model. These AssociationClasses are stereotyped with the «Structure2Analysis» stereotype so they can easily identified during the transformation process. A different correspondence is needed between every corresponding structure and analysis model. The relationship between the structural model of a double-acting cylinder and a corresponding analysis model is shown in Figure 4.8. These AssociationClasses are established by the designer for explicitly describing the relationships between a particular structural model and a particular analysis model. As there are a large number of potential connections between structural and analysis models, it would be beneficial to consider how computational support could be provided to automatically establish these connections, but this is left for future work.

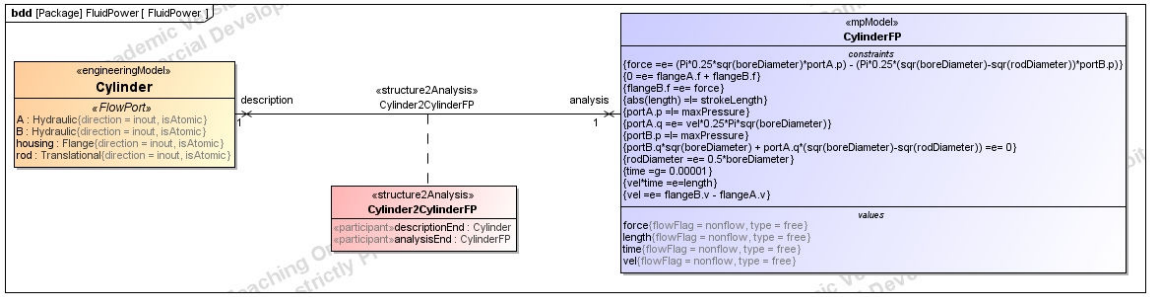

Figure 4.8: AssociationClass linking structural model of a double acting cylinder to an analysis model.

4.3.5 Parameter Maps

Model parameters from the component models are linked to parameters of the analysis model using binding connectors which are a standard construct of the SysML

language, the standard definition is they equate the two properties that have been connected together. They can also be combined with SysML constraints to capture algebraic relationships between the parameters. For example, a diameter in the structural model may need to be passed to an analysis model as a radius, so a constraint is used to specify this relationship. An example parameter map for the double acting cylinder is shown in Figure 4.9. Although some of the parameters of the structural model are linked one-to-one with parameters of the analysis model, this is not always the case. Since there are a large number of potential analysis models for a given structural model, there may be some parameters of the structural model that are not mapped to the analysis model. For example, parameters describing the thermal behavior of the system may not be considered when modeling the dynamics of the system.

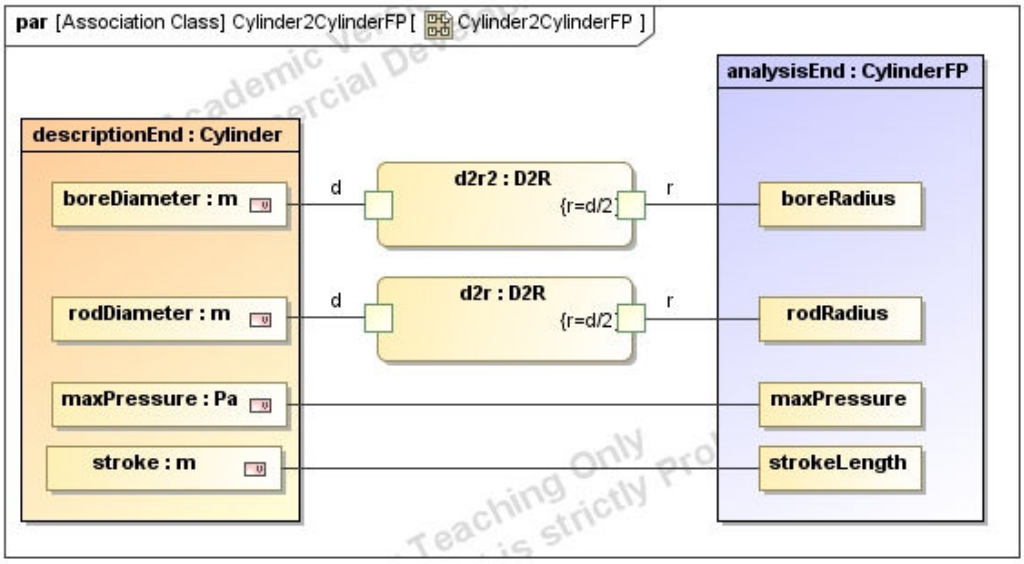

Figure 4.9: Parameter map between parameters of the double acting cylinder and the structural model.

4.3.6 Interface Maps

Just as parameter maps bind model parameters, interface maps are used to capture the mapping between the interfaces of the component and analysis models. The mapping between individual interfaces (modeled as ports within SysML) are also captured using connectors.

These fine grain connections between ports are established using SysML connectors as illustrated for the cylinder in Figure 4.10. These fine grain connections allow the creation of appropriate connections between analysis model interfaces.

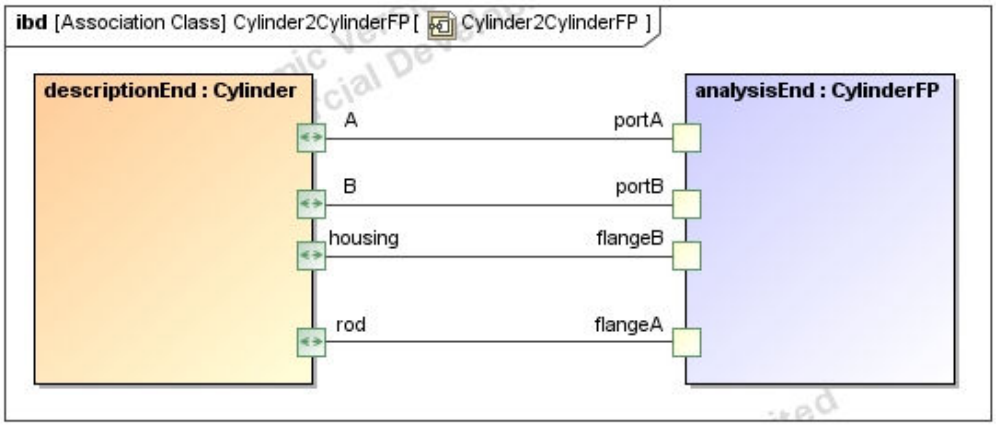

Figure 4.10: Interface map between the ports of the cylinder's analysis model and the structural model.

4.4 Automated Composition of Analysis Models

In this section, an approach is presented for composing analysis models with appropriate aspects from a representative model of the systems structure (whether the system is a physical system or not; this model will be referred to as a structural model) along with the knowledge captured within the model libraries. The approach relies on the

use of model transformations applied to the structural model to generate an appropriate analysis model.

In order to accomplish this, this transformation relies on the correspondences encoded as part of the model library described in Section 4.3.4. Without the models being represented in the same common language, defining these correspondences explicitly would be more difficult. Since these correspondences exist as explicit relationships, they can be traced during the composition process to relate the appropriate model fragments. Without these relationships, the transformation would have to rely on a more ad hoc and less flexible technique such as name matching to identify the appropriate parameters or interfaces. This composition approach can be used because the problem is being considered as being characterized by the composition of known components into more complex systems .

4.4.1 An Illustrative Example

The illustrative example involves the composition of an analysis model for the example system shown in Figure 4.11. The structural model is a system composed of the modular component (or subsystem) models, these models are either usages of models from the component (or subsystem) taxonomy or locally-redefined versions of those components. Locally-redefined versions are still specializations of models in the component taxonomy, so they inherit the appropriate relationships. When this specialization relationship is needed, SysML blocks representing the component models are linked to models in the taxonomy using SysML specialization relationships.

The models are connected via their interfaces, these connections are modeled as SysML connectors between SysML ports owned by the structural model; these

connections are maintained when the corresponding analysis model is generated. As an aside, it is not necessary for components in the structural model to be in the previously described taxonomy; the taxonomy has the advantage of facilitating component definition through the use of SysML's inheritance mechanism but components can be defined apart from the taxonomy. The only stipulation is that the components are related to the appropriate analysis models via AssociationClasses; if this is not the case, when the analysis model is composed they will not be included.

In the example, the circuit presented, illustrated in Figure 4.11 contains only a single pump, valve, cylinder and tank for its hydraulic subsystem. Also, it contains an engine to power the pump and a translational load that the cylinder actuates along with a controller for the valve. For example, the connector between the engine's "*flange*" port and the pump's "*rotational*" port represents a physical connection between the drive-shaft of the engine and the pump.

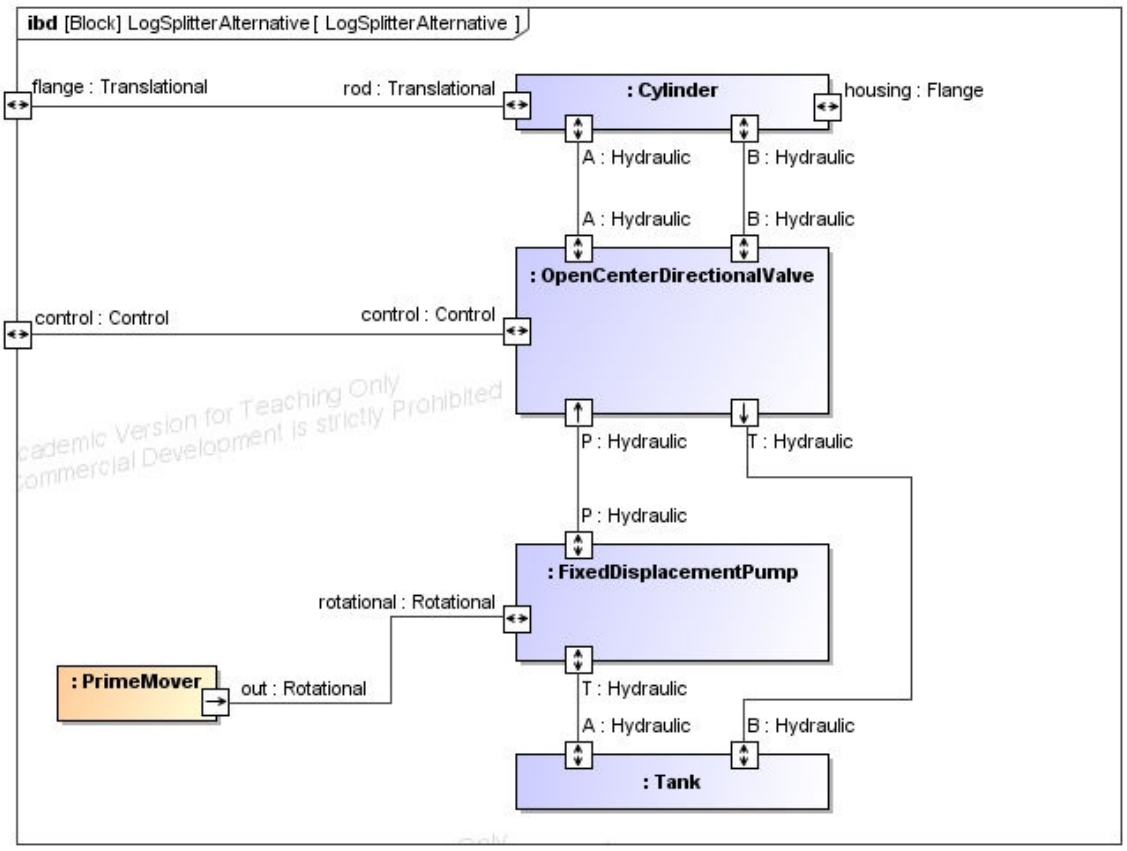

Figure 4.11: Example hydraulic circuit's structural model.

It is also important to capture exactly which analysis model should be composed from the defined structural model. In general, a single structural model may translate to a large number of possible analysis models. To capture this relationship between the structural model and the desired analysis model, a test context is used as illustrated in Figure 4.12. The test is associated with a set of aspects as well as the template structural model, the one shown in Figure 4.11. The definition of the test is based on the definition described in 3.4.1. When the corresponding system-level analysis model is composed; component-level models classified with the appropriate aspects are used based on the aspects corresponding to the test. The test can also prescribe the simulation parameters and specify the variables of interest as described previously.

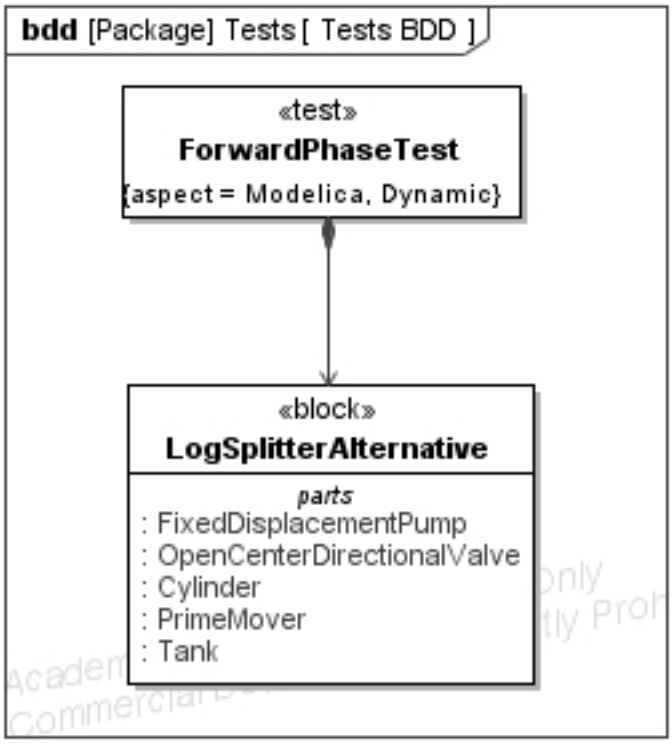

Figure 4.12: Test context describing the simulation and analysis model to be generated.

4.4.2 Creating the Analysis Model

Once component-level analysis models are constructed within SysML, model transformations are used to automatically compose them into system-level analyses and then transform them into executable simulations. Because a SysML model can be thought of as a labeled and directed graph, graph-based model transformations are used in this chapter. This transformation is broken up into two steps: first, from the simulation context into a set of system-level analysis models and second, from the set of analysis models into executable simulations. Once the analysis models are created in SysML, another transformation is used to create an executable simulation in a format compatible with a domain-specific modeling tool. As mentioned in the introduction, this chapter presents the transformation for a single system alternative, not for the full architecture

selection decision. The goal of the remaining sections in this chapter is to demonstrate the applicability and practicality of a composition approach. This serves as the basis for the more complete approach presented in Chapter 6. As a result, the transformation in Chapter 6 is presented in significantly more detail; here the goal is on presenting the general structure of the transformation and the artifacts that result.

In order to automatically create the model, a model transformation is used to *compose* the necessary analysis models from the system-level design alternative's structural model. Once the analysis model is created in SysML, it still needs to be transformed into a form that is interpretable by a specific tool. The first step is described in this section, while the second step is presented in the following section.

In order to compose the models, first the AssociationClass linking the structure and analysis view must be identified. Then a new usage of the component-level analysis model is included in the system-level model. Finally, based on the context of the component-level model in the structural view, composition relationships need to be instantiated in the system-level analysis model. This investigation identified three major types of composition:

1. Only the components present are important, such as with analysis models related to mass or cost, and the connections are irrelevant. Here, the appropriate attributes simple need to be appropriately aggregated.
2. Only the components and connections are important, such as in reliability chains. Here, the appropriate connections need to be instantiated but no additional information is needed.

3. The component and connections are important, but additional information is needed, for example in models where causality must be assigned or certain types of connections need to be replaced with nodes.

In its current form, the transformation only considers the first two categories. Additional research is needed in how to generally capture the additional information needed for the third case, although it could potentially be specified as meta-data related to the AssociationClasses, and is left for future work.

To simplify the definition, the transformation is decomposed into three distinct parts each applied to a different level of the structural model. The first transformation creates a new system-level analysis model that is consistent at the system level with the original structural model; i.e., the transformation creates a system-level analysis model that is composed of the models with the same component types present in the structural model. The second transformation maintains consistency at the component level; it creates the parameters and interfaces for each analysis model. The third transformation creates the appropriate connections between interfaces.

In previous work (Kerzhner, 2011), this transformation was implemented in a triple-graph grammar (TGG) styled approach where the transformations were defined with a meta-computer aided software engineering tool called MOFLON (Amelunxen, 2006). When the complete transformation is described along with the implementation in Chapter 6, it is done completely in Java. The basis for this change is discussed in that chapter.

Graph transformations are classically defined using a pre-condition, the part of the graph that is matched, and a post-condition, the replacement graph. For the system-level

transformation, the pre-condition is the structural model along with the appropriate model libraries.

The input to the transformation is a single model that includes the simulation context and any applicable model libraries. For each simulation context, a single system-level analysis model is constructed. To accomplish this, the transformation matches each component owned by the structural description of the system included in the simulation context. For each component, the transformation selects an appropriate component-level analysis model and adds it to the system-level analysis. The appropriate component-level analysis is selected by matching the aspects associated with the test to an analysis model with the appropriate aspects and correspondence relationship. This particular correspondence relationship relates the cylinder component with a model described the cylinder's behavior. Properties from the cylinder, such as the stroke, are equated to particular properties of the analysis.

The component-level transformation ensures consistency of component model parameters and interfaces. For the component-level transformation, the interface and parameter maps provide the majority of the information. This is first accomplished by replicating the parameters and interfaces of the analysis model in the library. The interfaces of the library models along with the previously mentioned parameter maps provide the templates for this transformation.

Once all the component-level analysis models are instantiated, they need to be correctly connected to other component-level models as well as to the inputs and outputs of the system-level analysis model. The first step is to instantiate new connections in the system-level analysis model for any connections between the interfaces of two

corresponding components. These connections are instantiated by using information contained in the «Structure2Analysis» AssociationClasses, which provide a correspondence between the structural and analytical models. Based on these relationships, the corresponding ports of the structural and analytical models are matched. If two ports in the structural model are connected, then that same connection is created in the analytical model by following the correspondence from both of the structural ports to the corresponding analytical ports. It is often the case that analysis models have additional parameters that are not present in the structural models. This can include initial conditions, simulation specific parameters, or other similar properties. These values are instantiated in the analysis models with their default values. Additional work may be needed to tweak these parameters based on the analysis model. A simple inheritance rule is defined to handle inherited properties being locally redefined.

Currently these transformations are applied in a batch-type operation; an entire system-level analysis model is composed through the application of the transformations. Future work will investigate how the use of correspondence objects will allow incremental updates of the system-level analysis model from modifications to the structural model.

There are several considerations when defining compositions between interfaces. In general, the assumption is that structural interfaces connected using SysML connectors correspond to connecting the interfaces of the analysis model with connectors. But, for several types of analyses this assumption does not hold. Simpler cases are easily included in this presented definition, if the analysis models being composed require only information about a models position or no connectivity information at all (for example

mass, moment of inertia) this is easily captured using the presented framework. Capturing compositions where additional structure is required, such as replacing connection configurations that result in interfaces having cardinality not equal to one with nodes forcing the interfaces to have a cardinality of one, is more difficult because these unique compositions need to be captured unambiguously. The implication is that this additional knowledge must either be included as part of the connection templates or as part of the transformation specification. Ideally, this knowledge would be encoded as composition rules using a generic language and included as part of the definition of the connection templates. This would require that the transformation is capable of interpreting these composition rules, which would make the transformation specification more difficult to create. If there are only a small number of composition rules that are unlikely to change, then it could be more practical to directly encode these as part of the transformation specfication.

The resulting analysis model is shown in Figure 4.13. This model is structurally similar to the model shown in Figure 4.11, but the structural models have been replaced with external analysis models. Also, one can see that the names of the ports for instance have changed, yet the correct connections are still present. This model can be mapped into Modelica concrete syntax to allow for simulation. Because Modelica is an object-oriented modeling language, the transformation into Modelica is straightforward: the appropriate class types need to be instantiated and then connected together via their interfaces. The code generator takes each part property in the composed model and generates code to instantiate the appropriate model type from the Modelica library. Then connect statements are generated based on the connectors present in the SysML model.

This resulting model is shown in Figure 4.15. A similar approach for doing discrete-event modeling in SysML has been demonstrated in Huang (Huang, 2007).

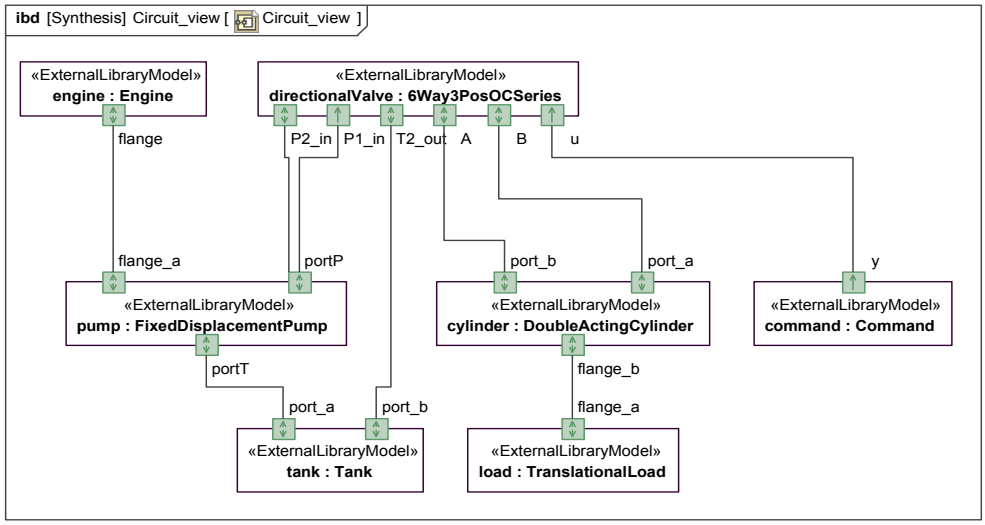

Figure 4.13: Resulting analysis model.

4.5 Referencing external models in a model library

In this case, only part of the component-level analysis models is represented in SysML. Most of the definition, such as the equations and internal workings, are represented in Modelica code which is captured external to the SysML authoring tool. To enable this separation of representation in SysML and model definition, additional elements are added to SysML to allow the encoding of so-called library models. This allows for the use of pre-existing models that have been defined outside of SysML, in this instance Modelica models that could come from the Modelica Standard Library. This has the advantage of referencing existing or legacy analysis models without requiring the manual effort of redefining these models within SysML.

Each model in the library is a “black box”; it references an existing model outside of the SysML tool. In order to create such a “black box” model and subsequently

reference an external model, several pieces of information are needed. These are captured within the «*ExternalLibrary*» and «*ExternalModel*» stereotypes. The «*ExternalLibrary*» stereotype requires the "*url*" tag where information pointing to the location of the library is stored. The «*ExternalModel*» stereotype requires the "*ref*" tag which stores information about the location of that particular model within the library. The stereotype also needs either the "*library*" tag which points to the associated library or a "*url*" tag. The definition of these stereotypes is shown in Figure 4.14.

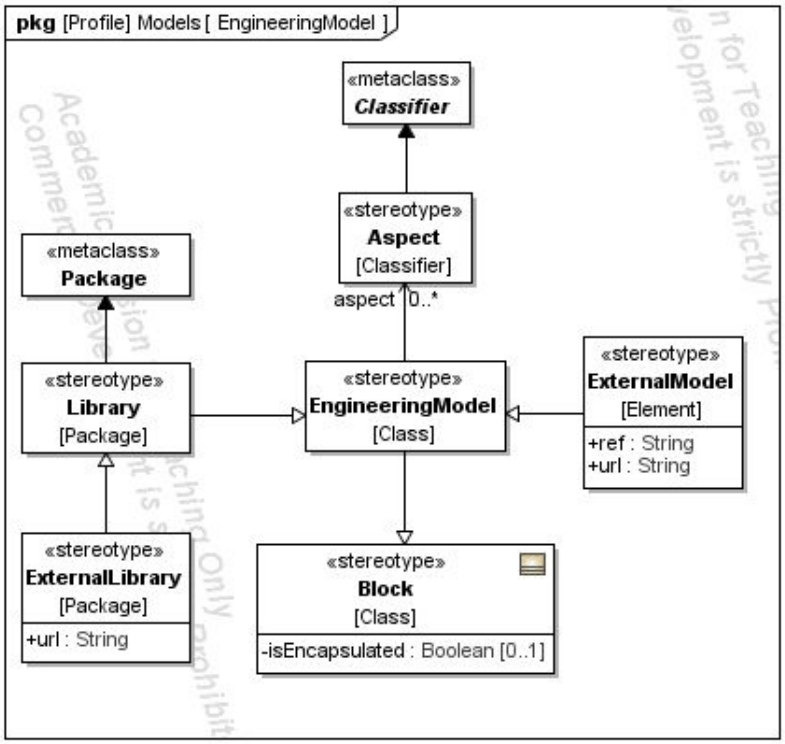

Figure 4.14: Profile for defining external models and libraries.

Using these external libraries along with a pretty-printer that goes from the SysML representation into Modelica code, executable Modelica code can be generated from the SysML analysis model. The pretty printer is based on the previous work (Johnson, 2012). This resulting model is illustrated in Figure 4.15.

```
model Circuit
 Modelica.Mechanics.Translational.SlidingMass load;
 Modelica.Mechniacs.Rotational.ConstantSpeed engine;
 Modelica.Blocks.Math.Sin command;
 FluidPower.Components.Cylinders.DoubleActingCylinder cylinder;
 FluidPower.Components.Valves.DirectionalValves.SV6_3OCSeries directionalValve;
 FluidPower.Components.MotorsPumps.ConstantDisplacementPump pump;
 FliudPower.Components.Volumse.CircuitTank tank;
equation
 connect(cylinder.flange_a, load.flange_a);
 connect(directionalValve.A, cylinder.port_a);
 connect(directionValve.B, cylinder.port_b);
 connect(directionalValve.P2_in, pump.portP);
 connect(directionalValve.P1_in, pump.portP);
 connect(command.y, directionalValve.u);
 connect(pump.PortT, tank.port_a);
 connect(directionalValve.T1_out, tank.port_b);
 connect(engine.flange, pump.flange_a);
end Circuit;
```

Figure 4.15: Resulting Modelica code.

4.6 Discussion

Although in this section only a single analysis model is considered, it is possible to apply the same approach for different types of analyses in different domains. For instance, from a single structural model, one could generate a cost model, a reliability model, a mass model, or an algebraic steady-state model. In general, defining the transformation rules for composing dynamic analysis models from structural models is non-trivial. A significant problem is the selection of causality assignments; that is: which of the variables describing the behavior of a component should be considered as inputs and outputs when combining the model with other component models? Through recent advances in symbolic manipulation of Differential-Algebraic Equations (DAE) (Beltrame, 2006), several simulation tools now support causality assignment (and even index reduction) in an automated fashion. These tools are based on declarative, object-oriented modeling languages such as Modelica (Mattsson, 1998), VHDL-AMS (Christen, 1999), or SimScape.

In addition to the automated composition of continuous dynamics models, it is possible to define similar transformations for discrete-event simulation models. Several hierarchical, object-oriented modeling languages and tools have been developed for discrete-event simulation (Garrido, 2001, Varga, 2008, Zeigler, 1987). Recently, Huang *et al.* (Huang, 2007), have already considered integrating discrete-event models into SysML, so that discrete-event models can be closely tied to corresponding descriptive models. In their approach, the SysML language has been extended using stereotypes to represent different types of manufacturing assets. These semantically-rich models contain all the information to convert a logical or structural description of a manufacturing line into a corresponding simulation model in eM-Plant (Heinicke, 2000).

Although there are clear benefits to automating the process of generating analysis models in an automated fashion, there also some costs associated with it. For instance, there is overhead in capturing both the analysis models and structural models formally within SysML. There is also overhead in linking these models together in a form that allows the automatic generation of system-level analysis models. Some of this overhead is mitigated because there is an opportunity for reuse of the models for future problems. Some of the opportunity for reuse comes from the modular nature in which the correspondences between analysis and structural models are captured independent of system-level considerations. Further research is necessary to evaluate carefully how the costs of defining the models in a more formal fashion tradeoff against the benefits of using these models at a much reduced cost.

One aspect that has not been considered is situations where additional knowledge is needed to compose the analysis model or where additional knowledge is needed to appropriately configure the simulation.

4.7 Summary

In this chapter, an approach for capturing reusable model fragments in model libraries is presented. The focus is specifically on reusing analysis models, although structural and analysis models are captured in libraries. Capturing reusable fragments does not add value to the modeling process without a practical approach to actually reusing these fragments. Having the structural models available has the additional benefit of simplifying the definition of the architecture selection decision as described in the previous chapter. In addition, the second part of this chapter presents a transformation approach for composing together component-level analysis models into system-level analysis models. Although in this chapter this is demonstrated for only one architecture alternative, in Chapter 6 this transformation approach is extended to the entire space of solutions.

CHAPTER 5:

ARCHITECTURE SELECTION USING MATHEMATICAL PROGRAMMING

In the previous two chapters, the focus was on representing an architecture exploration problem as an architecture selection decision, which includes both domain knowledge and a designer's intuition about the design space. Starting with this chapter, the focus shifts toward performing an architecture exploration process to guide designers in making an architecture selection decision. In this chapter the focus is on RQ 3:

RQ3. What mathematical framework is best suited for identifying promising architectures?

In this chapter, the focus is on how the knowledge that is needed for an architecture selection decision can be encoded in the mathematical programming domain. The goal is to lay the foundation for supporting H3 by demonstrating how the relevant knowledge can be encoded as a mathematical programming optimization problem, then Chapter 7 provides an illustrative example where mathematical programming optimization tools are applied to support an architecture exploration process. As a reminder, H3 is as follows:

H3: Designers could use mathematical programming techniques to identify promising solutions early in the exploration. Mixed-Integer Linear Programming should be used for architecture selection.

In addition, early on in the design process when the solution space is large and difficult to search, one way to improve the solution speed is to use mixed-integer linear programming; by utilizing a MIP representation instead of a MINLP representation should make the approach for scalable. The issue is that even early on in the design process when very inaccurate models are being used, nonlinear effects may have a significant impact both on feasibility and optimality. Therefore, in this chapter there is

also some discussion on how inherently nonlinear behavior can be approximated in a MIP representation.

The approach to modeling the architecture selection decision is presented in a modular framework for modeling architecture exploration problems. Within this framework, which components and connections can be included in a potential alternative and also analysis knowledge that can be used to identify feasible and promising alternatives is represented. Although using mixed-integer linear programming can be restrictive, by appropriately structuring the equations it is argued that the formulation is sufficient in early stages of architecture exploration.

The justification for using mathematical programming, specifically mixed-integer linear programming, is the availability of high-quality commercial solvers that can be applied to this problem. Because they are designed for large-scale global optimization, these solvers can take advantage of the structure of the problem to improve solution times and also provide some assurances about the quality of the solution. A linear formulation is selected instead of a nonlinear formulation because in general linear problems are easier to solve and the solvers are more robust. Previous experience with nonlinear solvers such as the Branch and Reduce Optimization Navigator (BARON) (Sahinidis, 1996) showed that if variables were not well-bounded, the solver could incorrectly characterize a feasible design space as infeasible (Shah, 2010c).

The rest of this chapter is outlined as follows. First, some desired characteristics for identified for an effective solution approach in Section 5.1. Once these characteristics have been delineated, some alternative approaches are considered from the perspective of the requirements in Section 5.2. After establishing the limitations with current

approaches, the mathematical programming-based approach is presented in Section 5.3. The goal of this chapter is to describe how a mathematical programming problem for a particular architecture selection decision can be composed. This process is considered composition because there is a clear mapping between certain parts of the architecture selection decision and resulting mathematical programming constructs. To construct the full mathematical programming problem, these different parts are composed into a single problem formulation as is described in Chapter 6.

5.1 Desired Characteristics of the Search Process

Although in Chapter 1, some desired characteristics for the overall method were presented, in this section the specific desired characteristics for the search process is described. These characteristics are derived from both the problem description presented in Chapter 3 and also general desired characteristics for common design processes. In part, these characteristics are supposed to combat the difficulties that arise when searching a space of a large number of potential system embodiments with different architectures.

Early in the design process, there are a large number of alternatives to consider; each alternative is the sized embodiment of a potential architecture. If one were to visualize the search space, it might look like Figure 5.1. In this figure, the space of architectures is shown on the left. For each architecture, there is a continuous space where sizing occurs, as illustrated on the right. If these two views were flattened into a single space, alternatives with the same architecture but different component sizings create continuous regions in the space, but alternatives with different architectures would appear in discrete regions. In order to define this design space both continuous and

discrete variables are needed. Although it would be desirable for the framework to handle both variable types, discrete variables are the most important because they are needed to differentiate between architectures. Continuous variables are still important but early on in the process the goal is to identify promising architectures and sizing is a secondary concern; also, continuous variables can be discretized into a set of discrete choices if needed although this can influence solution times.

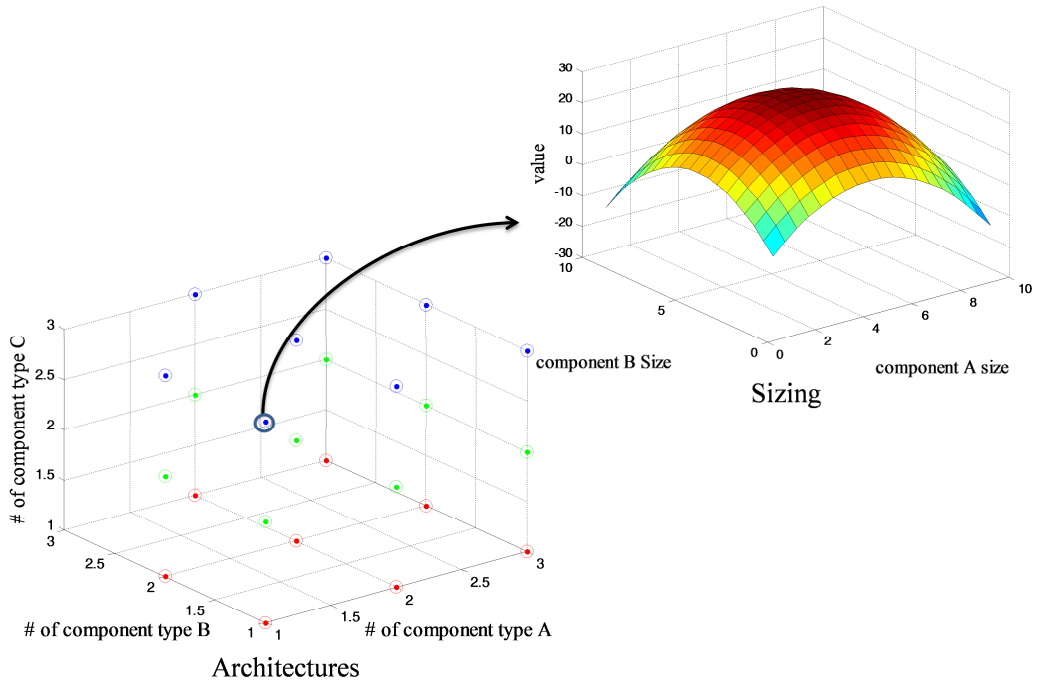

Figure 5.1: Visualization of the design space.

Because of current computational imitations, both in process speed and available memory, it is necessary to scope the exploration so that solutions can be found in a reasonable amount of time. This can be done in two ways, by reducing the number of potential solutions that are being considered or by reducing the accuracy of the analysis performed during the solution process. Although removing clearly inferior or infeasible solutions from the search space would help the search, the goal is to consider a wide

range of solutions. The more appealing approach is to reduce the accuracy of the analyses used to differentiate solutions. This factors into how domain knowledge is represented throughout the process. Early on, there is the potential to use a significant number of simplifying assumptions and represent the domain knowledge in a simplified form to reduce the complexity of the problem formulation, but simplifying assumptions reduce the chance that the best candidate architecture identified during early exploration is truly the optimal solution. For example, instead of evaluating systems based on their dynamic behavior and representing the problem as a set of differential equations, the steady state behavior can be used instead and the problem can be simplified into a set of algebraic equations.

Since it is unlikely that the solution to a simplified problem is truly the optimal solution, the solution framework must be able to generate multiple solutions which can be further explored. Since the goal is to identify several promising architectures, it is also important these solutions embody different architectures. There are many approaches to generating multiple solutions, for instance with a genetic algorithm approach, any feasible solution can be recorded in a solution pool and some fitness criteria can be used to eliminate solutions from this pool.

Another important factor is the time it takes for the solution process to identify promising solutions; it is important that any solution process can finish in a reasonable amount of time and not cause a bottleneck in the design process. Currently, one major factor that prevents designers from exploring a wide range of architectures is the limited time that designers are budgeted to work on a certain project.

Because an exploration is made up of a large alternative space, where many of the alternatives may be infeasible, such a search is usually very time consuming. Therefore, it is important to identify problem formulations and solution approaches that can handle the problem efficiently. Also, if the goal is to serve as a support tool for designers, it might be beneficial for designers to execute the exploration multiple times using different available components or objectives to perform trade-studies early in the process. Solution time is very important in enabling such tasks. One common approach to reducing the solution time is to employ parallelization. This requires additional computation resources, but reduces the amount of time designers wait for solutions.

Also, it is desirable to use knowledge about an architecture's behavior (even if it is at an abstract, functional level) early in the process. Creating metrics based purely on the structure of the architecture is difficult because generic mappings between an architectures structure and its performance are not readily available. Previous work has attempted to use artificial neural networks to perform classifications and differentiate architectures, but this proved to be difficult because of a lack of training data. There are a number of generic complexity metrics (Summers, 2010) which try to describe an architecture by the number and nature of components and connections, but such metrics make significant assumptions about the architectures behavior. Also, making generic statements like less "complex" architectures are always better is not always meaningful.

5.2 Choice of Solution Approach

In this section, an exploration of potential solution approaches is presented. Although rarely referred to as supporting architecture selection decisions, there are a number of previous approaches for identifying feasible architectures for a given problem

or context. Most of these methods are referred to as computational design synthesis approaches, with the focus on synthesizing feasible candidate architectures from ad hoc representations of the problem. A review of these approaches was provided in Section 2.3, the most relevant are again highlighted here. One important characteristic of this investigation that distinguishes it from previous approaches is that the problem formulation contains designer knowledge about synthesizing candidate solutions as well as analyzing and evaluating them; all three are included in a single comprehensive framework.

These previous approaches can be classified into either grammar-based approaches or constraint-based approaches. Graph grammars provide a formal language for specifying the design space (Mullins, 1991b), but often this design space is specified in an ad hoc manner. They have been successfully used in a number of applications, but the transformations can be difficult to define (Bolognini, 2007, Starling, 2005). Although the data structures and transformations approaches can be based on custom code, computer-aided software engineering tools have been used recently to simplify their specification (Amelunxen, 2006, Fischer, 1998). Also, grammars are usually only used for creating topologies and a completely separate approach is used to solve for component parameters. One notable exception is attribute grammars (Mullins, 1991a) where configuration and parametric design is considered a part of the grammar. The drawback of using graph transformations is that there is a need for a large number of transformations that must be specified manually. Insuring that the result of a transformation is still within the space of alternatives is also difficult and requires the transformations to be defined very precisely (Wyatt, 2012).

In the grammatical approaches, generating an architecture usually involves a set of transformations and the evaluation of a particular architecture is completely separate from synthesizing that architecture. This makes it significantly more difficult to guide the solver because it is difficult to characterize how changes to which transformations are applied effects the resulting architecture. Since grammatical approaches only provide a representation of the design space, a separate search approach is needed to evaluate solutions and explore the space. A common approach is to use some form of a genetic algorithm or some other stochastic search algorithm and consider the knowledge encoded in the grammar and related analyses as a black box.

Genetic Algorithms (GAs) (Goldberg, 1989, Holland, 1992) are a very common technique for searching discrete spaces. In a genetic algorithm, a fitness function is calculated but the analyses that are executed to evaluate this fitness function are treated as a black-box. Instead of considering only a single candidate solution, GAs rely on a population of solutions. New solutions are created by modifying this population; commonly mutation or cross-over operations are applied to the population of solutions to generate new candidate solutions. Then some selection criteria is used (most often a Russian roulette style approach) to choose which solutions are included in the solution pool for the next iteration (the pool is commonly referred to as a generation).

The problem with classical GAs is that the search space is defined as a set of binary variables with generic cross-over and mutation operators. For certain optimization problems (specifically when searching for feasible architectures), most of the solutions resulting from these classic cross-over or mutation operations are not feasible or very poor making the search process inefficient.

To address this issue, there are a number techniques which include domain knowledge about the search space (or the search domain) in the representation of solutions or in the cross-over and mutation operators; genetic algorithms that include this additional domain knowledge are often referred to as evolutionary programs. Evolutionary programs have been shown to generate high quality solutions in a number of fields, for example electric circuit design (Koza, 2001). Some form of GAs is often used with grammar-based approaches, especially those relying on graph transformations (Emmerich, 2001). The potential solution is represented as a graph (which is a natural representation for a system architecture) and the mutation and cross-over operations can be implement as graph transformations.

GAs are a global stochastic optimization technique that are largely problem independent, although as mentioned in the previous section domain knowledge can be included to make them more efficient for certain problems. GAs can be used on discrete search spaces without well-defined distance metrics because only the fitness function is needed to compare solutions. Genetic algorithms do not attempt to understand the relationship between inputs and the objective function; instead the cross-over and mutation operators are applied at random. Because of this inherent randomness, genetic algorithms should converge over time to the global optima(s) of a continuous or discrete search space as long as some sequence of mutation or cross-over operations will allow the algorithm to move between any two points in the space. Issues can arise when the algorithm is not able to move from one feasible solution to another because of a highly constrained space or ineffective mutation operators. In that case, each generation may remain in the same general regions and not truly explore the space. This can partially be

offset by starting with a large randomly generation initial population, but this slows the speed of the search process.

Most other methods (and the approach presented in this investigation) can be classified as constraint-based approaches; usually the constraints are specified as either a set of equations or using a custom constraint language. Previous approaches have demonstrated the applicability of mathematical programming approaches in this area, with Biegler et al. demonstrating the automatic synthesis of chemical reactor networks (Biegler, 1997, Yeomans, 1999). The chemical network is represented as a superstructure which is a union of all possible alternatives; it is a conglomeration of all potential architectural options. Decision variables are used to represent which options of a superstructure are included in a particular alternative. Constraints are then added to specify which sets of options specify valid alternatives and to specify the expected behavior of a particular alternative. Wyatt et al. describe the structure of the system using a small set of constraints, and then synthesize alternatives by modifying an existing solution and checking the feasibility of the new solution using the constraints. The constraints are placed directly on the system structure, and alternatives are evaluated purely on their structure (Wyatt, 2012).

In this investigation, a mixed-integer linear programming optimization problem is used to represent the architecture selection decision and IBM's CPLEX (International Business Machines Corp, 2009) is used to solve it. CPLEX is a mixed-integer linear programming optimizer; it has been extensively used in the mathematical programming domain. The search process employed by CPLEX can be generically described as follows. It uses a branch and bound approach for handling the binary variables; in this

approach is creates a search tree whose nodes include different combinations of the binary variables. For some of these nodes, a relaxed linear programming problem is solved. The results of this relaxed problem are used to guide the addition and deletion of nodes on the search tree.

The use of mathematical programming techniques is similar to the chemical network approach, although the scope of the formulation presented here is much more complete. In the chemical network examples, the structure of the network is fixed with only several optional components. Here, the entire structure of the architecture can be variable. Also, the formulation described here can be defined in a modular fashion which supports the automatic generation of this formulation from a generic problem definition. As mentioned earlier, the major difference is in the scale of the problem. Since a constraint satisfaction-based approach is chosen for this investigation, the next sections describe the characteristics of constraint satisfaction approaches and also further support the choice of mathematical programming.

5.2.1 Constraint Satisfaction Approaches

In a constraint satisfaction (CSP) formulation, the definition of the objective function and accompanying constraints are no longer a black-box. Instead, they are codified in a way that can be interpreted by the solver so that the solver can consider the structure of the constraints and manipulate these constraints into simpler problems with the knowledge gained being used to guide the search process. This makes this class of approaches particularly attractive for use during an architecture exploration process because of the promise to efficiently explore very large spaces.

The other advantage of constraint satisfaction approaches is that the definition of the problem is separated from the solution mechanism. In many previously mentioned approaches, the algorithm operated on a black-box with pre-defined inputs and outputs; in order to perform an architecture exploration using these approaches, custom analyses are needed that appropriately map inputs to outputs, making the approach difficult in practice. With a constraint satisfaction approach, the constraint satisfaction solver can "perform" the relevant analyses as long as these analyses can be formulated in the language of the solver.

In this section, several constraint satisfaction approaches are recognized that may be appropriate for the solution process. These approaches are compared qualitatively and the rationale for selecting mathematical programming, specifically mixed-integer linear programming is presented. This selection is based on the current state of the art; in the future, a hybrid method that mixes solution techniques commonly used in mathematical programming with solution techniques from other constraint satisfaction approaches may provide the best results (Yunes, 2010).

Constraint satisfaction approaches can be categorized by the types of variables and constraints that can be handled by the solver. Some solvers can handle only continuous variables, while others can handle only discrete variables while others can consider both. Also, the structure of the constraints used varies from logical statements to purely linear equalities and inequalities to nonlinear equalities and inequalities. There are a number of different constraint satisfaction approaches and a more complete survey can be found in previous work by Kumar (Kumar, 1992).

Previous work has used constraint satisfaction approaches to perform engineering analyses, usually to select appropriate sizing parameters for a fixed architecture. Constraints are used to define the feasible combinations of the sizing parameters and to capture an objective function which is maximized (minimized) by the solver. For example, Decision Support Problems (DSPs) provide a number of templates for modeling common designer decisions in this form (Bascaran, 1989). Although there are many different types of DSPs, the most common is the compromise DPP (cDSP) where the objective is to minimize a weighted average deviation of the solution from prescribed targets. In this investigation, the concept is extended to not only include sizing parameters but also variables that describe the structure of the architecture.

5.2.2 Boolean Satisfaction

Boolean satisfaction is a class of constraint satisfaction approaches where only Boolean variables are considered and logical statements constrain these Boolean variables. The solver's goal is to then identify a set or sets of Boolean variables that do not invalidate any of these constraints.

The logical statements are often based on either first-order or descriptive logic. To give the reader a better idea about the nature of these logical statements, a brief overview of first-order logic follows. In this investigation, one of the justifications for choosing mathematical programming (with algebraic constraints and continuous variables) over an approach based on a logical system (with logical constraints and discrete variables) is the ease in which designers can represent their analysis knowledge.

First-order logic (FOL) (Smullyan, 1995) is a formal logical system which is more general than propositional logic or descriptive logic. FOL allows for formulas

constructed with four types of symbols: constants, variables, functions, and predicates. Descriptive logic is a subset of first-order logic.

- Constants represent the objects in the domain of interest.
- Predicates represent relationships between objects in the domain and can be either true or false.
- Functions represent a mapping from objects to objects
- Variables represent an instance of a particular type of constant.

An atom is a predicate symbol applied to a tuple of terms. Atoms are used to construct more complex formulates using logical connectives and quantifiers such as conjunction ($\wedge$, and), disjunction (v, or), implication ($\Rightarrow$), equivalence ($\Leftrightarrow$), universal quantification ($\forall$), and existential quantification ($\exists$).

Formulas can be converted to a conjunctive normal form where they can be rewritten as algebraic constraints in an integer programming formulation (Blair, 1986a). Therefore, mathematical programming solvers can also be used to solve problems involving first-order logic.

A major drawback of a Boolean satisfaction formulation is that even a single spurious statement can eliminate all possible solutions. Identifying spurious constraints is often very difficult and when no solution can be found, one cannot simply look at which constraints are violated or cannot be met, because in trying to satisfy the spurious constraints the solver may violate a number of correct constraints.

One way to address this problem is to use probability constraints in a formulation such as Markov Logic Networks (Domingos, 2008a). Here, instead of specifying only a set of constraints, weights are added to differentiate between constraints. As a result, a

solver is given guidance on which constraints are important and which can be violated; therefore a set of inconsistent constraints no longer eliminate all possible solutions.

Another drawback of Boolean satisfaction approaches is the lack of support for continuous variables. As a result, any analysis of the systems performance requires significant simplification to allow the analysis to be represented as a set of logical statements. This is a significant issue because engineering knowledge is rarely captured in logical statements; therefore, domain experts would need to take significant effort to capture their knowledge in this form. Also, with only logical constraints it is difficult to differentiate between different architectures. One way to address this issue is to allow the solver to find a large number of potential solutions; another approach would be to include additional constraints based on knowledge gained later in the search process.

In order to add these constraints automatically, the solution process could use machine learning techniques to identify common patterns in promising architectures and add these as constraints to the problem definition. The difficulty with using machine learning on this type of problem is the lack of training data.

5.2.3 Mathematical Programming

To address the drawbacks of Boolean satisfaction, one avenue is to move toward mathematical programming. The mathematical programming domain includes a wide variety of problem types which can handle both discrete and continuous variables. Instead of requiring additional analyses to provide an evaluation of a candidate architecture, continuous variables with algebraic constraints can be utilized for this purpose and included as part of the problem definition. Mathematical programming tools provide a number of high-quality commercial solvers. Also, there exist a number of

modeling languages for expressing mathematical programming problems in a form that is independent of any particular solver such as the General Algebraic Modeling System (GAMS) (Brooke, 1998), Another Mathematical Programming Language (AMPL) (Fourer, 1990b) or AIMMS (Bisschop, 2006). Therefore, the same formulation can be tested with multiple solvers. For example, with only slight modification a particular problem formulation can change from using only linear constraints to a non-linear formulation which uses a different solver and also better approximates the systems performance. In this investigation, the AIMMS modeling system is chosen, but the approach would be similar with only slight modifications if GAMS or AMPL were used instead. AIMMS is chosen because it currently provides a better user-interface and more debugging tools. The problem with using these languages is they are not well suited to engineering applications.

The most general form of mathematical programming is mixed-integer nonlinear programming (MINLP) where the problem is defined as follows:

$$\min Z = f(x, y)$$

$$s.t. \;\; g(x, y) \leq 0,$$

$$x \in \mathbb{Z}^n, y \in \mathbb{R}^p.$$

The functions *f* and *g* map *x* and *y* into $\mathbb{R}$ and $\mathbb{R}^m$ respectively. The advantage of representing design synthesis problems in MINLP form is the flexibility of the representation along with the ability to solve the problem using sophisticated, existing algorithms. Also, the definition of the problem is separated from the solution approach making it easier to explore the performance of different solvers on the specific examples considered in this research. There are a number of techniques to solve these types of

problems, such as brand and bound, cutting-plane methods, and reduction techniques (Tawarmalani, 2004). Although MINLP provides the most flexibility, MINLP problems are the most difficult to solve. If the architecture exploration can be represented using simpler equations, then there is an opportunity for faster solution times and improved solution results. In this investigation, the goal is to formulate a simplified version of the architecture exploration problem as a Mixed-Integer Linear Programming (MIP) problem. Instead of including nonlinear equations, the modeling of the systems behavior is simplified so that only linear equations are used. This is done in part because of the difference in robustness between linear and nonlinear solvers. When considering nonlinear problems, it is common that several solvers may find a feasible solution while others may not (Lastusilta, 2007). Since part of the goal of this investigation is to establish that mathematical programming optimization tools can be used on large exploration problems, it was decided that the more robust nature and better solution speeds of linear solvers made them more desirable.

Using MIP has several advantages and disadvantages. The main disadvantage is that the problem needs to be represented in a language that can be interpreted by a MIP solver. Also, since the goal is to use existing commercial solvers, there is not an opportunity as part of this investigation to tweak the optimization algorithms to this type of problem. For a MIP solver to be able to handle the architecture exploration problem, the problem must be represented in a set of integer and continuous variables and purely linear algebraic constraints. When nonlinear or dynamic behavior is considered, it will need to be approximated using linear algebraic constraints. The advantage of employing

MIP is the availability of a number of high quality solvers and a number of very efficient techniques to solve MIP problems.

5.2.4 Agent-Based Approaches

As an aside, there is an opportunity to parallelize many of the presented approaches across multiple computational nodes. Some of the processes can be directly parallelized, for instance constraint satisfaction solvers that employ a search tree mechanism can split this search process over multiple nodes and explore different parts of the tree in parallel (as is the case with CPLEX). Another common approach is to break up the tasks during the solution process and allocate these tasks to multiple agents each running on a different compute node. Others have used agent-based approaches to search the design space (Agarwal, 1999), specifically for simple electromechanical systems (Campbell, 2000). By employing independent computational agents, design synthesis algorithms can be decomposed and distributed across multiple computers. If these agents are considered as models of individual members of the design team, agent-based approaches can be used for design exploration. Agents usually have different roles, such as adding or subtracting components or evaluating an alternative. The drawback of agent-based approaches that use elementary agents in that they are very inefficient.

5.3 Structure of an Architecture Exploration Problem in MIP

Mixed-Integer Linear Programming (MIP) is rarely used in the context of architecture exploration in systems engineering. There is previous work by Biegler et al. (Biegler, 1997) in using mathematical programming techniques to support the design of chemical networks. These problem formulations included both an optimization of the architecture and an optimization of the sizing variables. Discrete variables were used to

describe the structure of a potential alternative; the set of discrete variables covered all potential solutions and a particular alternative is one particular solution combination. The mathematical programming problems created were relatively small, consisting of approximately 20 total variables. Because of the size of the formulation, nonlinear equations were included and the chemical networks had only a single usage phase.

These small problem formulations demonstrated the applicability of mathematical programming techniques for small architecture exploration problems, but do not provide a framework for formulating and performing architecture exploration processes on more complex systems. In order to extend mathematical programming into the broader domain of systems engineering, it is important to investigate how well mathematical programming problem definitions will scale and how well existing solvers will perform on these problems. Whereas the chemical networks had a largely fixed structure with only a few possible component configurations (in the examples provided, approximately 10), performing an architecture exploration requires considering systems with thousands of feasible configurations. Also, there is the added complication that these systems must perform adequately in a number of different use cases. These characteristics result in mathematical programming formulations that are very large, often including thousands of variables and constraints.

In order to formulate more complex mathematical programming problems, it is important to use concepts from object-oriented modeling (Paredis, 2001) such as modularity. In this section, the goal is to identify common structural features of a system architecture selection decision which can be clearly partitioned, describe how those features can be captured in a generic way within the mathematical programming

modeling language, and then compose these fragments into a single problem formulation that a solver can operate on.

In mathematical programming, the potential solution domain is explicitly defined in a set of variables before the solution process begins. Since this definition needs to happen initially, the size of the mathematical programming problem is potentially very large. This is unlike modification based approaches (most grammatical approaches) where a current solution is modified through a set of transformations (for example, mutation operations in an evolutionary program) to generate a new solution; the entire space of solutions does not need to be represented and instead only a set of current solutions needs to be stored along with the set of allowable transformations. The tradeoff is that if the search space is explicitly defined a priori, the solver then has a clear definition of this space and can make assertions about how thoroughly this space has been searched when returning a solution.

In addition to capturing the space of solutions, the mathematical programming problem needs to include the analysis knowledge to evaluate a particular solution. This knowledge is represented as a set of variables and algebraic constraints; these constraints need to be satisfied in order for a solution to be feasible.

Further, one of these variables can represent the selection criterion which is used to evaluate and rank solutions. Based on the selection of variables which describe the architecture, certain constraints will either be applicable or not included.

Sometimes, it is appropriate to differentiate between different kinds of constraints. While some truly define the feasible solution space, others constrain intermediate variables to support calculating the objective. These constraints can be considered lazy

constraints because they are usually trivially true. Identifying these constraints can improve the solution process.

In this investigation, a mathematical programming formulation of an architecture selection decision includes:

- Potential components and connections represented by a set of binary variables. These binary variables represent all potential components and connections.
- The sizings of all potential components described using a set of both continuous and discrete variables. This set of variables is usually continuous but discrete variables are used if off-the-shelf components are being used.
- The static performance attributes, for instance the mass or cost, of all potential components captured as a set of continuous and discrete variables. In addition, some variables are included to capture the aggregation of these component-level variables into a single system-level variable.
- The behavior or dynamic performance of all potential components represented using a set of continuous and discrete variables. Each variable definition is actually a set of indexed variables that allow the values to change with time or use scenario.
- To restrict feasible combinations of components directly, a set of constraints applied directly to combinations of binary variables. These can be considered logical constraints on the structure of the architecture.
- To further restrict the feasible combinations by considering behavior, a set of constraints describing the relationships between the behavioral variables. These indirectly describe feasible combinations of binary variables.

- The use scenarios and evaluations criteria captured with a set of variables and constraints.

This definition of the problem is more comprehensive than previous approaches.

5.4 Defining the Structure

The first step is defining the set of binary variables that express the space of architecture. Looking back to Section 5.3, these binary variables represent whether certain components or connections exist in the architecture. For instance, a particular variable will represent whether a specific connection between a certain pump's port and a certain valve's port exists. Instead of referring to a single connection or a single component, a particular variable may refer to a set of connections and/or components that are commonly used together; this reduces the overall number of variables and simplifies the problem.

This approach to defining the architecture is based on the superstructure approach where binary decision variables are used to represent possible system alternatives (Grossmann, 2002). In previous superstructure approaches, these binary variables are used to describe which components are included in a particular alternative. The connections between the components are defined statically based on whether a particular component exists or not. In this approach, these variables are used to represent the entire structure of the system which includes both the components in the system as well as how they are connected together via their interfaces.

Once these variables are defined, constraints are needed to define the feasible combinations of these variables. There are two different aspects to this. First, FOL statements are used to describe which combinations of binary variables represent valid

architectures by defining relationships strictly between these variables. Additionally, algebraic constraints are added to describe the system's performance in qualitative terms. These constraints also dictate which combinations of variables constitute a feasible architecture, but do so indirectly. The inclusion of system behavior, even in a very abstracted form, separates this approach from many others that use only constraints on the systems structure.

The FOL statements have the effect of specifying a domain-specific language for valid architectures, much like a metamodel or ontology. In order to represent these FOL statements within the mathematical programming problem, they need to be converted into linear constraints. First, the statements are converted into a normal form and then into a set of linear constraints. For instance, $A \rightarrow B$ (A implies B) in normal form is $\sim A \vee B$ which becomes $(1 - A) + B \geq 1$ as a linear constraint where A and B are binary variables which can be either 0 or 1. To full description of such conversions can be found in (Blair, 1986b).

5.4.1 Describing System Behavior

Once the structural description is established, the next step is to model the system's behavior. To accomplish this, additional algebraic constraints are added that include both the binary decision variables described in the previous section along with continuous variables that represent behavior of a particular component. In addition, algebraic constraints are added to describe potential connections in the system architecture. These constraints are only active when the appropriate components or connections are considered as part of the solution, i.e. the appropriate binary variables

have a value of one. More on these so-called optional constraints is presented in Section 5.4.2.

These algebraic constraints cannot predict how the system will behave without some knowledge of different usage scenarios. To describe usage scenarios, additional constraints are imposed on the boundary of the system. For the hydraulic excavator example, this includes specifying desired force and velocity produced by the systems actuators at several points in time; these usage scenarios are actually also a part of the tests described in Section 3.4.1. The implied assumption is that any solution that the solver finds to be feasible will be able to accomplish the desired use scenarios. This acts as a de facto low-accuracy screening process that eliminates solutions which are not capable of even performing the tests.

The algebraic constraints related to the system behavior can be defined in a number of ways. As described earlier, algebraic constraints are defined at a component (or subsystem) level. These algebraic constraints are supplemented with constraints that describe the connections between these components or subsystems.

Before these constraints can be specified, the appropriate variables are needed. For each potential component in the system, there are several sets of variables. Some describe the size of the component, for the pump it's mass, cost, or displacement. These variables can be picked by the solver but also need to be constant throughout the use scenarios. On the other hand, there are variables that describe the dynamic behavior, for instance in the pump there is a pressure differential that changes with different scenarios. Each of these variables is actually a set of multiple individual variables, one for each scenario. Because most mathematical programming languages are not object oriented,

each of these variables need to be defined separately for each component. Most mathematical programming languages do allow for quick definition of certain sets of variables. In this approach, variables from different components that capture the same quantity, for instance a components mass, are grouped into such sets. This also allows the quick definition of certain operations over these sets, for instance the summation of mass variables into a single system-level mass variable.

To be able to compose the components in this fashion, they are "connected" via well-defined interfaces (Paredis, 2001). Each component has additional variables that describe the interfaces of the component. Because of the structure of mathematical programming languages, the interfaces themselves are not modeled, instead only the related variables are captured. For the pump, that would include sets of variables that describe the pressure and flow at the intake and outtake ports along with variables that describe the torque and velocity at the pump's input shaft.

Algebraic constraints are then used to relate these variables. These constraints are the same for each component of a particular type and can be defined modularly and copied for every instance of the component.

A similar approach is used for connectors. Kirchhoff's Laws are used to describe the connections between components so that the component connections can be modeled. Each interface is described as defining exactly two variables: an across variable that describes the effort across the interface and a through variable that describes the flow through the interface. For instance, in the mechanical domain, the across variable is velocity and the through variable is force. This across and through formulation is common in many simulation languages, such as Modelica (Fritzson, 2004).

Connections between interfaces then translate into a set of algebraic equations which describe how the across and through variables relate to each other. The advantage of using this approach is it is independent of the actual across and through variables. As long as the variables can be distinguished, the source domain and their individual nature are not important. Kirchhoff's Laws state at a node the across variables should be equated and the through variables should sum to zero. Here, the interface is viewed as the node.

For each interface, the across variable is equated to the across variable of any other connected interface. Therefore, for each interface there is a set of equations for the across variable; this set of equations is the same size as the number of connections to the interface. Each equation takes the form:

$$p_{\mathrm{A}} = p_{\mathrm{B}}$$

where p_{A} and p_{B} represent the across variable at interface A and interface B respectively.

In order to define the relationships between through variables, additional variables are needed; it is not possible to use only the previously defined interface variables. For each connection an additional variable is added to describe the flow of the through variable through that connection. At each interface, only a single equation is needed where the connector flow variables are summed with the interface's through variable and equated to zero. This equation takes the form:

$$Q_{\mathrm{A}} + Q_{\mathrm{AB}} + Q_{\mathrm{AC}} = 0$$

where Q_{A} is the through variable of the interface (the flow into the interface) and Q_{AB} is the flow from interface A to interface B and Q_{AC} is the flow from Interface A to interface C. At interface B or C, these variables would also be present but with the opposite sign, i.e.:

$$Q_{\mathrm{C}} - Q_{\mathrm{AC}} = 0$$

In other domains where Kirchhoff's laws might be inappropriate, other composition rules can be used as long as the composition can be represented as algebraic equations.

5.4.2 Optional Constraints

Although specifying the connection equations in this form is sufficient for connections that will always be present in a solution alternative, some connections in the problem definition are only included in a specific alternative. There are several ways to handle this. In classic mathematical programming approaches, a big M formulation can be used to approximate the if-then statements needed (Lee, 2011). For example, a statement such as if a certain connection exists in this architecture, than this constraint must be true. The problem with the big M formulation is it requires careful selection of upper and lower bounds.

Some solvers, specifically CPLEX, have built-in optional constraints, often referred to as indicator constraints. These are used in this formulation to handle any optional constraints; in the big M formulation bounds need to be intelligently chosen when formulating an if-then statement. By using indicator constraints, CPLEX manages the formulation automatically reducing the burden on the designer (or the transformation approach). For each potential connection, an additional constraint is added as follows:

$$Q_{AB} = 0$$

where Q_{AB} is the flow from interface A to interface B. An indicator constraint is added to specify that this constraint is active only when the connection does not exist. In addition,

an indicator constraint is included with the across constraint to specify that relationship is only active when the connection exists.

Similarly, indicator constraints are also used in conjunction with equations which are only applicable when the component is included in the formulation. For example, if a component is not included in the formulation, a similar set of constraints is added which describes that no flow can enter or leave the component.

5.4.3 Interpolation

Restricting the formulation to purely linear equations can be difficult, especially if there is nonlinear behavior that is important in the problem. For instance, in the excavator example in Chapter 7, the fuel consumption is approximately based on the power used by the engine, to calculate the power the product of torque and angular velocity is needed. To handle this situation, piecewise interpolation is used to approximate the nonlinear behavior; this allows the nonlinear behavior to be represented in a set of mixed-integer linear constraints.

Two different types of interpolation are presented here: a generic method to approximate 1-dimensional nonlinear functions and a generic method to approximate products between two variables. These two methods can be used in conjunction to approximate more complex functions; also, the concepts described here can be extended to create more complex interpellants. These approximations are common tricks in the mathematical programming community (Bisschop, 2006) but here the focus is specifically on when and how they should be applied when describing the problem.

One-dimensional interpolation requires as a design input only a data set of input and output value combinations. For an unknown variable y which is a function of x, i.e. y=F(x), the interpellant is as follows:

$$y = y_1\lambda_1 + y_2\lambda_2 + \cdots + y_n\lambda_n$$

$$x = x_1\lambda_1 + x_2\lambda_2 + \cdots + \mathrm{x_n}\lambda_\mathrm{n}$$

$$1 = \lambda_1 + \lambda_2 + \cdots + \lambda_n \qquad (1)$$

where $\mathrm{x}_{1..n}$ are the given inputs and $\mathrm{y}_{1..n}$ are the corresponding outputs and $\lambda_{1..n}$ are unknown and chosen by the solver. The other constraint is that only 2 adjacent λs can be nonzero; in most mathematical programming languages this can be represented specifying that equation 1 is a special order set (SOS) 2 condition.

To better understand this interpellant, an example is shown in Figure 5.2. The red curve represents the true function, the points the set of input data. The interpolation is represented by the black line segments, with a particular interpolated value is highlighted with the red star.

This interpellant can approximate both convex and non-convex nonlinear constraints. For approximation of purely convex constraints, the SOS 2 condition is not needed and the interpolation can be made up of only linear constraints with no binary variables. For non-convex linear constraints, the SOS 2 condition is required. The SOS 2 condition can be thought of as adding binary variables to the formulation of the interpellant so that the interpellant includes both linear constraints and these integer variables. This is an interpolation that is often used in the operations research community, and CPLEX can automatically handle the SOS 2 condition.

The SOS2 constraint insures that the interpolation remains on the line segment bounds. If this constraint is left out, then the interpolated value can be anywhere within the shaded region. It should be clear from the figure that that more input/output combinations that are provided, the more accurate the interpellant, the trade-off is that the inclusion of additional variables makes the problem more difficult to solve.

If instead of an equality constraint, an inequality constraint is being interpolated, there is the chance that the SOS2 constraint is not needed. As can be seen in the in the example figure, as long as the function is convex and one is interested a less than case, or the function is concave and one is interested in a greater than case, then the SOS2 constraint can be excluded. This slightly simplifies the set of equations used which can improve solve time.

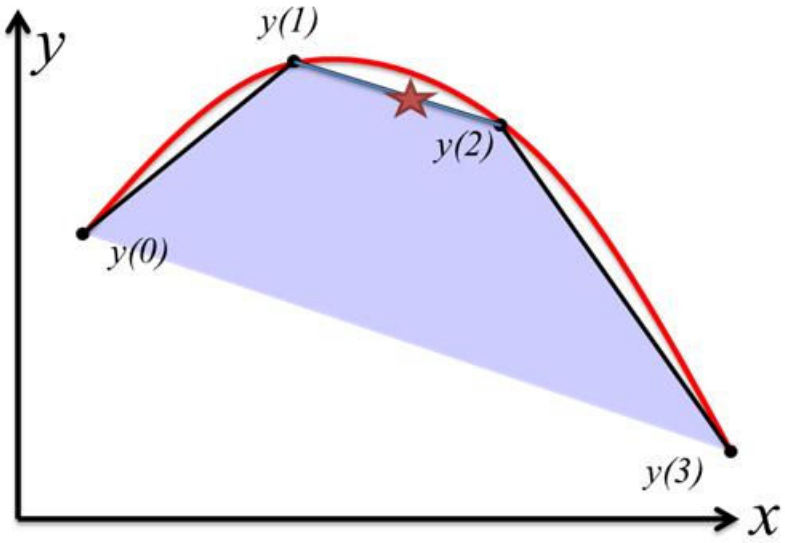

Figure 5.2: One dimensional interpolation.

To interpolate a product, a different formulation is used. This formulation is more complex because it needs to accommodate two input variables.

For an unknown variable z which is a product of x and y, i.e. $z = x \cdot y$, the interpellant is as follows:

$$a = \frac{1}{2}(x + y)$$

$$b = \frac{1}{2}(x - y)$$

$$z = a^2 - b^2$$

where a and b are intermediate variables and the values of a^2 and b^2 are approximated using the previous interpellant:

$$a^2 \cong a_1^2\lambda_{11} + a_2^2\lambda_{12} + \cdots + a_n^2\lambda_{1n}$$

$$b^2 \cong b_1^2\lambda_{21} + b_2^2\lambda_{22} + \cdots + b_n^2\lambda_{2n}$$

$$a = a_1\lambda_{11} + a_2\lambda_{12} + \cdots + a_n\lambda_{1n}$$

$$b = b_1\lambda_{11} + b_2\lambda_{12} + \cdots + b_n\lambda_{1n}$$

Using these various interpellants allows the approximation of most nonlinear constraints. By including additional input-output combinations in interpellant, the nonlinear function is approximated more accurately. The tradeoff is that the inclusion of additional points also requires the inclusion of additional λ variables which must be chosen by the solver. Another important factor revolves around the nonlinear constraint being approximated. If it is an inequality constraint, then the SOS 2 constraint placed on equation 1 can be relaxed reducing the difficulty of solving the problem.

The issue with this second interpellant is scaling; accurately computing $a^2 - b^2$ when the magnitudes of x and y are significantly different requires either a large number of interpolation points or for the points to be intelligently selected. Using a large number of points is intractable because this also results in a large number of lambda variables

which the solver must accurately choose. As an example, in Figure 5.3 an interpolation of $f(x, y) = x \cdot y$ where x is between 0 and 0.01 and y is between 0 and 10 is shown with both $a^2$and b^2being interpolated with 5 points each; these points are evenly spaced between $a_{min}$and a_{max} and $b_{min}$and b_{max} The red points represent the true value of $x \cdot y$ where as the surface represents the interpolated values. Although intuitively it makes more sense to visualize the interpellant as a set of points and the true data as a surface, representing the interpellant as a surface in this case allows it to be visualized more clearly. As expected, near the choice of input/output combinations, the interpellant is fairly accurate (as reflected by the surface nearing the points), but the accuracy quickly decreases away from these sampled points.

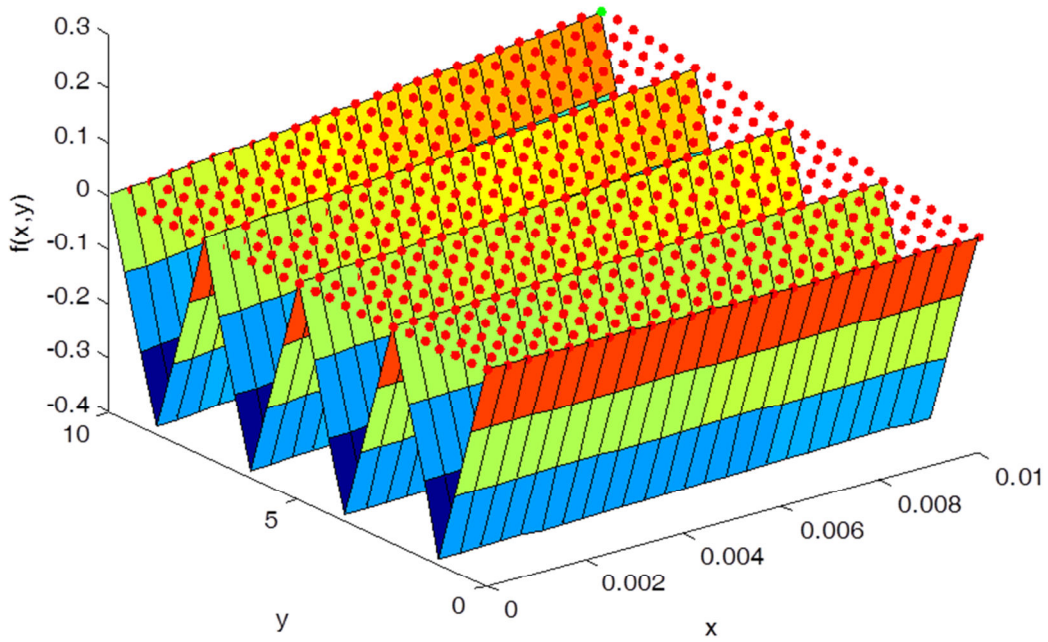

Figure 5.3: Unscaled interpellant. The red points are true data and the surface is the interpellant.

This can be a significant issue in engineering problems where it is common for variables of significantly different magnitudes to be multiplied. To provide an example from the hydraulic systems domain, when computing the force at the cylinders it is common to multiply the a cylinder area (the rod or bore side) which is usually less than .1 m^2 with the pressure applied across that surface which is usually on the order of 10^7 Pa to find the force in N. Since the common magnitudes of these variables are known and it is usually possible to express upper and lower bounds for these variables, one way to address this issue is to scale *x* and *y* so the magnitudes are similar. Also, by centering the *a* and *b* variables around zero, the symmetric nature of squares can be used to cut the number of points needed for the same accuracy by almost half. To implement this scaling, the *x* and *y* variables are scaled to be between -1 and 1. As a result, the values of *a* and *b* are between -1 and 1. The scaling is performed as follows:

$$x_s = 2\frac{x - x_{min}}{(x_{max} - x_{\min})} - 1$$

$$y_s = 2\frac{y - y_{min}}{(y_{max} - y_{\min})} - 1$$

To unscale the result and find $f(x, y) = x \cdot y$ from $x_s \cdot y_s$the following is used:

$$f(x,y) = \left(\frac{1}{4}(x_s \cdot y_s) + \frac{1}{2}x_s + \frac{1}{2}y_s + \frac{1}{4}\right) \cdot (x_{max} - x_{min}) \cdot (y_{max} - y_{min}) + \frac{1}{2}(x_s + 1)$$

$$\cdot (x_{max} - x_{min}) \cdot y_{min} + \frac{1}{2}(y_s + 1) \cdot (y_{max} - y_{min}) \cdot x_{min} + \ y_{min} \cdot x_{min}$$

where $x_s \cdot y_s$ is approximated using the interpellant and the rest are known values. To compare, in Figure 5.4, the left plot shows the original unscaled result, on the right the result of the scaled interpellant.

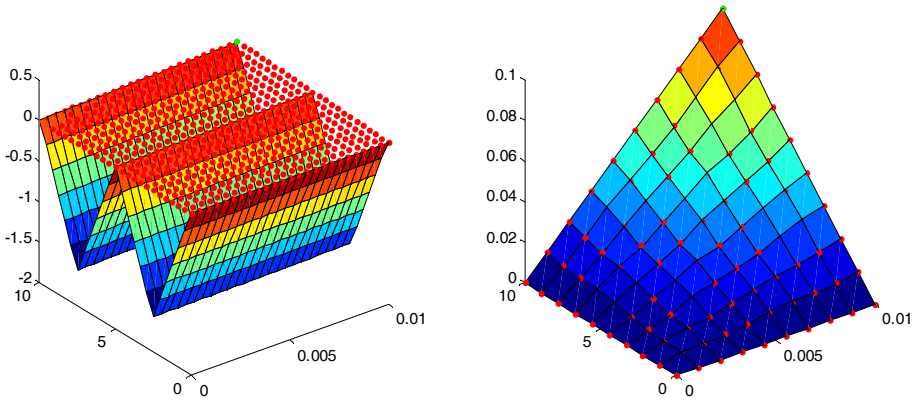

Figure 5.4: Unscaled versus scaled interpellants.

These two interpellants are sufficient for any constraints that can be represented as a set of functions. On the other hand, if the nonlinear behavior being approximated is that of a lookup table, the one-dimensional interpellant may need to be extended to multiple dimensions. This is a trivial extension, although the number of λ variables is equal to the number of data points used. In multi-dimensional cases, this can lead to a very large number of variables which can significantly impact the solution time of the solver. Therefore, if possible it is desirable to represent such interpolations as a set of the 1-D interpellant and the interpellation of products.

5.5 Representing Algebraic Analysis Models in SysML

In order to enable the transformation approach presented in the next chapter, it is necessary to also represent the algebraic component models within SysML. Stereotyped SysML constraint blocks can serve this purpose, along with constraints, properties, and ports. The modeling approach is based on prior work by Shah *et al.* (Shah, 2010c) where General Algebraic Modeling System (GAMS) models were stored in an object-oriented form within SysML. In this prior work, a profile was defined for representing the GAMS

models; this profile is generalized in this work to include algebraic models generically. This is possible because of the similarities in the modeling languages used between the different mathematical programming modeling tools, such as AIMMS, GAMS, and Another Mathematical Programming Language (AMPL). The profile used is shown in Figure 5.5; the «*MPModel*» stereotype is used to identify mathematical programming models. These models can consist of constraints, ports, and properties. Properties can be stereotyped as «*MPVariables*» which transfer into variables. Constraints also have several stereotypes that can be applied. Three stereotypes are of particular interest here, the «*InterpConstraint*» stereotype for representing constraints that need to be interpolated using 1-D interpolation when linear programming is used. This stereotype allows the inclusion of data points with the constraint which specify the interpolation. Also, if these data points are not fixed but instead depend on the component being modeled, this can also be specified. In that case, these values will come from values found in the structural component library described in Chapter 4. In addition is the «*MultiplicationConstraint*» which specifies constraints that need to be approximated using the multiplication interpolation presented in the previous section. Also, the «*MPConditionalConstraint*» allows the specification of constraints that are only sometimes active. This is used in the specification of valves for instance, where there are multiple potential use phases and different equations associated with each phase.

By providing a language for representing algebraic analysis models in SysML, designers and domain experts can encode their domain knowledge and then use composition to reuse this domain knowledge across multiple projects. The addition of stereotypes to facilitate the transformation of nonlinear constraints into linear constraints

during the transformation process described in Chapter 6 also reduces the effort required to capture these models An important precursor to model reuse is to create the necessary model libraries and verify that they are accurate. One issue with the creation of analysis model libraries is that in order for the analysis models to be reusable the assumptions and context must by similar or generic. A generic library model may include a number of user-specified options that change the nature of the equations used in the model and therefore the related assumptions. When comparing to the current state of practice, the creation of commercially available model libraries is common. One instance is Simulink, where the tool-vendor (MathWorks) provides the model libraries to make their tool easier to use (Mathworks, 2008). Another instance is the Modelica community, where the model libraries are created by community members. Some of these members are companies that sell the model libraries (Modelica Association, 2012).

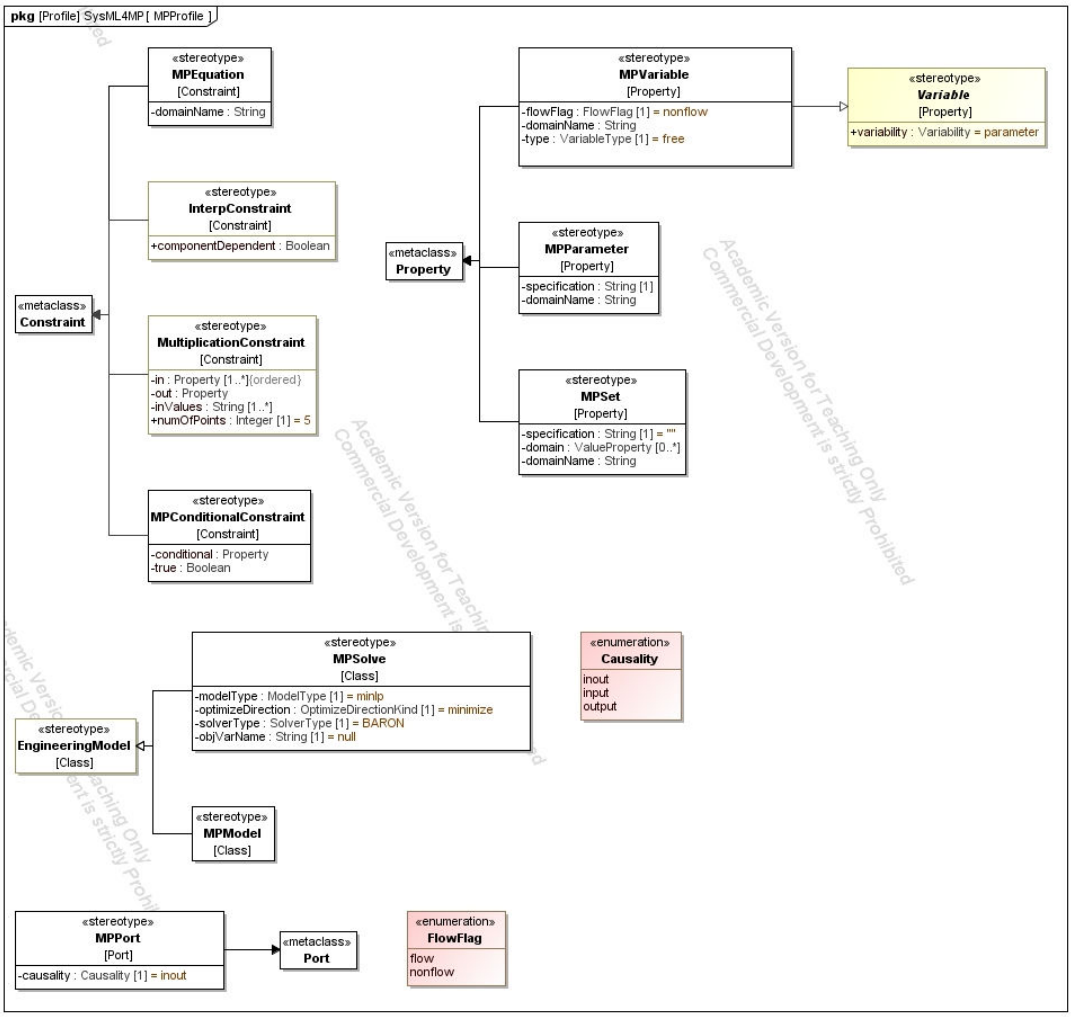

Figure 5.5: Profile for representing algebraic models in SysML

5.6 Discussion

Here, the approach is to formulate an architecture selection decision as a mathematical programming problem. The problem is simplified by restricting all the constraints to linear constraints. This choice to use linear constraints is supported by the argument that using only linear constraints makes the problem easier to solve. Although this is often the case, because much of the system's behavior is approximated using interpolation which introduces additional variables, it is possible that using the original

nonlinear equations with a mixed-integer nonlinear solver which could potentially have better results. This tradeoff will be explored more closely in future work.

The other consideration is generating this mathematical programming representation. As will be seen in the example problem in Chapter 7, asking human designers to take the time and effort to accurately and efficiently create large mathematical programming problems, interrupting the design process to do so, is not practical. This is one of the major barriers of using mathematical programming approaches in the current state of the art. In this chapter, a foundation for model transformations is provided by decomposing the architecture selection problem modularly, and describing how each piece of the problem maps into the mathematical programming domain. These described mappings are used as the basis of the transformation in the next chapter. The key enabling feature is considering the selection problem as involving the composition of well-defined components into more complex systems. By providing a clear mapping between the problem definition and the mathematical formulation, model transformations can be applied to automatically generate the problem.

Previous approaches have shown that model transformations are capable of transforming semantically rich information models into a number of different analysis formulations (Czarnecki, 2006). By employing these techniques, using a mathematical programming approach is no longer impractical. This allows the opportunity for significant experimentation of this approach, and allows engineers working in this area access to a number of high-quality commercial solvers which appear well suited to

support this type of decision making but are currently not used for the reasons previously described.

5.7 Summary

In this chapter, a framework for describing an architecture selection decision within a mathematical programming problem is presented. The framework is demonstrated on the selection of an actuation subsystem for an excavator. An important characteristic of this approach is the selection decision is represented modularly which simplifies the problem definition. For each type of element found in the problem definition, the mapping into the mathematical programming formulation is described. By providing a structured approach to generating this mathematical programming representation, model transformations can be applied to automatically generate a specific formulation for a specific problem description.

The goal of this chapter is to provide support for H3 by demonstrating how the same concepts behind the SysML representation can be represented in mathematical programming. In the next chapter, a transformation between these two representations will be presented because although it is possible to manually create the formulation as described in this chapter, for even small architecture selection decisions it is not practical. One of the enabling aspects of the transformation is the so-called modular approach in which the mathematical programming constructs are used. This allows the transformation to identify the appropriate part of the SysML model and compose the mathematical programming representation by converting small sections of the model into the appropriate code.

CHAPTER 6:

PROBLEM TRANSFORMATION

In this chapter, a transformation approach from the problem definition described in Chapters 3 and 4 to a corresponding mathematical programming problem as described in Chapter 5 is presented.

The aim of this section is multi-fold. The first is to provide further infrastructure for addressing RQ3 and RQ4 because an approach is needed to quickly generate number of different executable problem statements from the same problem definition. Recall:

RQ3. What mathematical framework is best suited for identifying promising architectures?

RQ4. How should problem scale be managed?

The second is to further illustrate the value of explicitly defining the architecture selection decision as described in Chapter 3 by demonstrating that the mathematical programming problem can be automatically composed from the definition. Automatically creating this representation has several potential advantages, although there is some additional overhead in explicitly modeling the architecture selection decision:

1. Mitigation of the additional effort required to explicitly model the architecture selection decision through the reduction of non-value added effort resulting from duplication of design knowledge between the SysML model and the mathematical programming representation.
2. The ability to model and generate larger mathematical programming models for architecture selection than is possible with current state of the art mathematical programming systems.

3. Increased opportunity for reuse of knowledge between different architecture selection decisions.
4. Opportunity to perform error detection and consistency checking on the object-oriented SysML model.

There is the alternative of manually creating the mathematical programming representation by referring the SysML formulation. As will be illustrated in this chapter and in the examples, it is not desirable for designers or engineers to create these representations directly because of their sheer scale and the opportunity for error. Although to truly validate this statement, user studies are needed, in this investigation only a logical argument is developed based on the quality of available tools and the size of the models.

There are several alternatives in implementing this transformation based approach. These will be discussed in the next section, Section 6.1. Then, the structural differences between the object-oriented model represented in SysML and the flattened mathematical programming representation are highlighted along with other transformation issues in Section 6.2; these structural differences significantly increase the complexity of the transformation process. Then the transformation process is presented with discussion on how it addresses these issues in Section 6.3. The transformation is accomplished in two stages, in the first stage most of the structural differences are resolved in a new model and in the second stage, presented in Section 6.4, this new model is used to pretty-print code that is interpretable by AIMMS. Because most mathematical programming systems use similar languages to define mathematical programming problems, this second stage can be easily modified to target a different tool.

6.1 Defining Model Transformations

The automated transformation procedure presented in this chapter is implemented using Java code. There are a number of model transformation approaches for defining this transformation, many of which promise the explicit modeling of the transformation specification making it easier to review and maintain. As the majority of this investigation has favored explicit modeling, it is contradictory to not use model transformation approaches to define and execute this transformation. The justification for representing the transformation directly in Java stems from the quality and maturity of existing model transformation tools and approaches along with the ease in which these approaches can interface with the chosen SysML authoring tool, in this case NoMagic's MagicDraw UML (MD) (No Magic Inc., 2012). MD provides an extensive Java interface that can interface with a number of Eclipse based tools such as an implementation of the Object Management Group's Query/View/Transform (QVT) standard.

In general, model transformation approaches can be generically described as transforming between models, with the actual specification of the transformation occurring at the metamodel level. A metamodel describes the possible structure of models that conform to it; it defines the potential constructs of the modeling language along with the potential relationships between these constructs. An alternative view of a metamodel is that it defines a modeling language that other models can use. This generic description is illustrated in Figure 6.1. In the future, model transformations are expected to play an important role in MBSE for modeling and implementing interfaces between the plethora of tools used by designers when designing a system (Stahl, 2006).

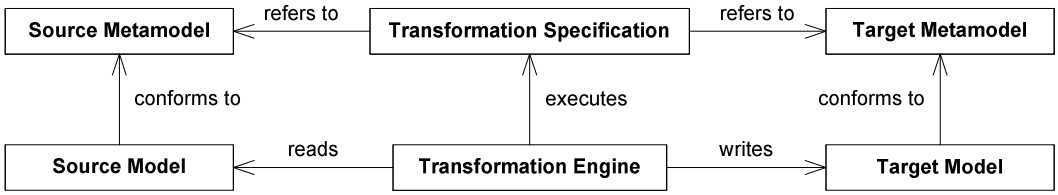

Figure 6.1: Generic structure of model transformations. Adapted from Czarnecki (Czarnecki, 2006).

Current applications of model transformations include model synchronization and the generation of low-level models/code from high-level models. Many methods exist for implementing these transformations; two common approaches are OMG's Queries/Views/Transformations (QVT) (Object Management Group, 2007) and Triple Graph Grammars (TGGs) (Schürr, 1994).

The QVT specification provides a set of languages for querying a source model that complies with a source metamodel and transforming it into a target model that complies with a target metamodel. Two QVT languages, *Relations* and *Core*, are used to model declaratively the relationships between the source and target metamodels. The *Operational Mappings* language is then used to describe imperative transformations based on the relationships depicted in the *Core* or *Relations* languages. The relations between the QVT languages are depicted Figure 6.2. Since QVT is an OMG standard, the definition of the language is both a standard and comprehensive. The issue is in the tools implementing the language.

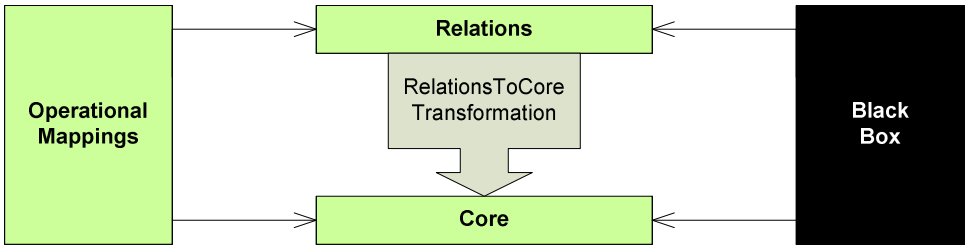

Figure 6.2: Relations between the QVT languages (Object Management Group, 2007).

Overall, QVT is a widely accepted model transformation language; however, there is a lack of tool support to execute the language. Also, until very recently, the integration between MD and these QVT tools has been difficult. In the latest version of MD there is a built-in integration for operational QVT, but this capability was not considered as part of this investigation.

TGGs are similar to QVT in intent but are declarative by nature. Accordingly, TGGs are particularly useful for completing complex, bidirectional model transformations; however, others have shown that QVT is equally expressive and capable (Greenyer, 2007). In a TGG, two modeling languages (metamodels) are defined as graphs. The mapping between the two metamodels is then represented by an intermediary graph called the *correspondence metamodel*. This third graph is essential for defining graph transformation rules and maintaining traceability links between the two models. A practical implementation of TGGs is also demonstrated extensively by Königs (Königs, 2006). The issue with TGGs is they significantly constrain the types of transformation rules that can be specified. They are extremely effective in areas where the mapping is one-to-one, but they remain difficult to apply when significant modification of the structure is needed.

6.2 Transformation Issues

As mentioned in the introduction, the structure of a mathematical programming problem is significantly different than the structure of a SysML model; this makes the transformation between the two significantly more difficult than if they were structurally similar. In the OMG's SysML-Modelica specification (Paredis, 2010), one of the simplifying factors was the object-oriented nature of both SysML and the Modelica language. The flattening, pretty printing, and compiling that occurs to transform a model specified in the Modelica language into executable simulations is handled by the individual Modelica tools (Åkesson, 2010b).

Although a mathematical programming tool does do some post-processing on the defined mathematical programming problem before it is interpreted by a solver, the provided mathematical programming language (from AIMMS and also other tools) is completely flat with a relatively small number of constructs specifically geared toward describing a set of variables and constraints. The modeling language is described as flat because there is only one namespace, elements cannot be organized in a hierarchical fashion, and common object-oriented concepts like inheritance or redefinition are not available.

To simplify the transformation, the SysML model could be restricted to look structurally similar to the mathematical programming formulation or only a few object-oriented concepts could be included in the transformation process (Shah, 2010c). This is also similar to the s-COMMA GUI approach where mathematical programs are represented in a simple object-oriented language similar to UML (Chenouard, 2008). From the representation presented in Chapter 3, it should be clear that the goal of this

investigation is not simply to represent mathematical programming problems in an object-oriented fashion. Instead, the goal is to provide designers with a convenient language to represent their architecture selection decisions and automatically transform that representation into a form where an architecture exploration search process can be executed. Attempting to force this language to be structurally similar to the mathematical programming domain would require engineers to explicitly model many of the constraints that are automatically handled in a generic fashion in this approach. Åkesson demonstrated an approach for generating flattened nonlinear mathematical programming optimization problems for AMPL from a more abstract and object-oriented Modelica representation of the analysis (Åkesson, 2010a). This suggests that performing the flattening process as part of a transformation is feasible.

The additional difficulty is that mathematical programming provides only a few types of constructs and, unlike in SysML, these constructs cannot be easily extended. Therefore, the knowledge present in the SysML model must be mapped onto only the available constructs. This is partially the goal of the work presented in the previous chapter, where the focus was on how to represent various aspects of the architecture selection decision in the mathematical programming domain.

An additional consideration that arises from these structural differences is that translating knowledge from a mathematical programming model into the SysML representation is difficult. This is not considered here because it is unlikely that an engineer or designer would rather represent his or her knowledge in the mathematical programming domain because of the significant restrictions. In real-world applications, there may be some instances where large mathematical programming problems already

exist, but it is likely that if these problems are represented modularly like the architecture selection decision they would be significantly smaller in the SysML formulation. Still, an importer could be built to import some of the equations into SysML structural elements, but this is left for future work.

6.2.1 Practical Considerations

The fact that both the SysML representation and the mathematical programming representation are derived from the same structuring of an architecture selection decision enables the definition of the transformation. In addition, there are some practical considerations which must be taken into account when performing the transformation. Because of the lack of hierarchical structure in the mathematical programming domain, it is important to eliminate potential name collisions. In SysML, because of the hierarchical structure there are multiple namespaces (a container for a set of identifiers, names). In mathematical programming languages, the flat structure does not contain multiple namespaces which introduces the possibility for elements having the same identifier resulting in name collisions. Also, many mathematical programming languages impose limits on the length of names so often times a hashing process is needed to replace the names used in the SysML model with shorter placeholders in the mathematical programming problem. This also can make it difficult to identify the instance-level mapping between the SysML and the mathematical programming formulation; this correspondence between the original constructs in SysML and the renamed constructs needs to be explicitly captured. In addition to the renaming, because the description of a particular architecture is represented as a set of binary variables in the MIP problem, it is difficult for a designer to visualize the result of the optimization. A process is needed to transform the set of values returned by the optimizer into architecture description that can

be understood by the designer. Therefore, there is also the need to translate the results of the mathematical programming problem back into SysML (the approach taken as part of this investigation as will be seen in the next chapter) or into another form where they can be easily visualized and reviewed by the designer.

In instances where there are multiple usages of the same type in the SysML model, the definition of the type must be copied separately each time in the mathematical programming formulation or indexed constraints must be used. When there are multiple components of the same type included in the problem definition, in the SysML model the type is only defined once and then each usage references the original type. In the mathematical programming definition, the definition of the type must be separate for each usage, which means that designers must include copies of the type definition within the problem formulation or create additional sets related to the usages and use indexed variables and constraints. Also, some structural elements in SysML that implicitly represent variables or constraints to enhance usability need to be transformed into explicit definitions of those variables or constraints. For example, the binary variables related to the selection of a particular architecture are implicitly defined through the use of connection templates and multiplicities in the original SysML model. The definition of the problem only implicitly defines these variables, so the transformation process needs to identify these variables and make them explicit. Another option would be to adjust the problem definition to explicitly define the variables, but this would make defining, modifying, and reviewing the problem more cumbersome.

Also, because most mathematical programming languages do not support inheritance, any inheritance and redefinition in the SysML model must be resolved into a

flattened form where inheritance or redefinition does not appear. This is done by explicitly defining any inherited or redefined constructs from the SysML model when representing them in the mathematical programming language, similar to the flattening process that occurs with Modelica (Åkesson, 2008, 2010b). Resolving this inheritance is one of the most difficult aspects of the transformation because the usage and resolution of the inheritance and redefinition constructs within SysML is still not standardized. As a result, there are a large number of different use cases the transformation must take into account.

The need to flatten the very complex structure of the original model requires that a large number of correspondences are maintained between the new and original model. Maintaining these correspondences between the SysML model and textual code is very difficult. Considering the correspondences implicitly or trying to capture the correspondences using low-level coding concepts such as hash or tree maps is also difficult.

Therefore, in this transformation approach an intermediate model is created in SysML with a structure similar to the mathematical programming code that needs to be generated. The correspondence between this model and the original problem definition can then be explicitly captured. In addition, during the definition of the transformation process this intermediate model is more easily reviewable to provide confidence that the transformation is performing as expected.

This intermediate model has some other important benefits which make a two-stage transformation process the more desirable than the one-stage approach. The most important aspects center around how easy the transformation is to create, review,

maintain, and modify. By generating an intermediate model in SysML, the bulk of structural transformations can be reviewed, often visually, without need to parse large amounts of resulting textual code. Also, the same intermediate model could be used to generate different types of executable models such as either linear or nonlinear mathematical programming modes. Also, the intermediate model can serve as the basis for visualizing the results because it contains the same variables that are present in the mathematical programming problem along with the relationship between those variables and the structural features in SysML.

6.3 Transformation Process – First Stage

The transformation is performed in two distinct steps beginning with a SysML model and ending with code that can be executed by AIMMS. Although the final goal is to generate code that can be interpreted by a mathematical programming modeling system (in this case, AIMMS), an intermediate step is taken to generate a flattened model that is structural similar to the resulting code. This flattened model is also captured in SysML but is simply a bi-product of the transformation. This simplifies the code generation process because no structural changes are needed. Instead, the code generator (pretty printer) can simply print the appropriate equations for each element in the intermediate model. The first transformation step is the most difficult because the in the transformation requires a significant structural change to the model.

To help illustrate the transformation, a very simple partial model fragment is used. The SysML definition of this fragment is shown in Figure 6.3. Because it is only a model fragment, it is difficult to describe it as an architecture selection decision. Instead, the main aspect captured is that the potential circuit contains a power unit (which includes a

pump and tank) and a valve block (which includes an open-centered valve along with a check valve). The reason this particular example is included is to highlight how the object-oriented structure of the SysML model is transformed into a flattened structure and finally to AIMMS code. Since this is a model fragment the resulting AIMMS code that is created during the transformation will also be incomplete.

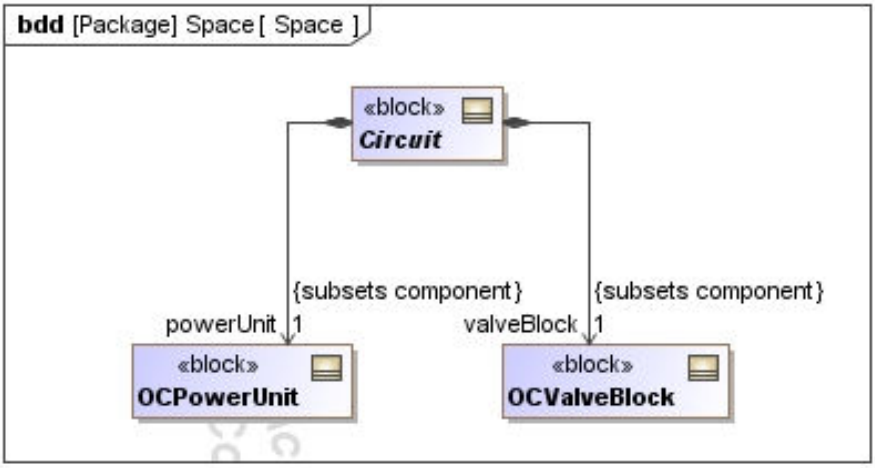

Figure 6.3: The structural definition of a simplified model fragment

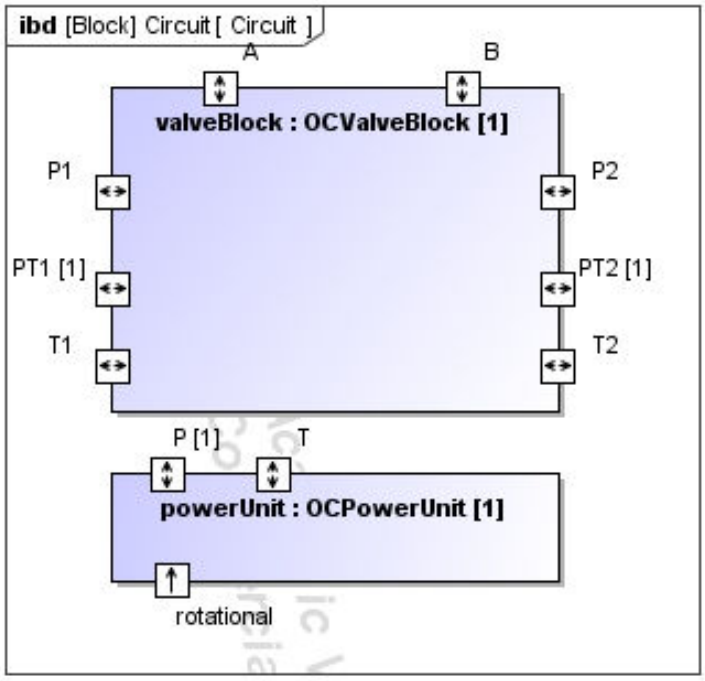

Figure 6.4: The internal definition of the simplified model fragment

The intermediate model created by the first transformation step is really a *superstructure* of the definition of the selection decision. A superstructure is a union of all potential architectures with all possible connections and components present (Biegler, 1997), as was discussed in the previous chapter. In addition, the components included in any tests are also in the superstructure. Not only does the superstructure contain all possible components and connections, but it includes them in a flattened form separate from the original problem definition.

The elements contained within the superstructure are specializations of the original model elements. In addition, these elements contain a copy of the appropriate subset of variables and constraints from the relevant elements in various analysis libraries. Also, additional variables are added to describe a single potential architecture in the superstructure.

The first step to generating the superstructure is to identify all the potential components. First the structure of the abstract system (in this case the abstract circuit block) is traversed (the transformation starts at the root of the hierarchical model structure and searches this structure using a depth-first search) and the concrete atomic components (or types) are identified for inclusion in the super structure; the definition of an abstract system was described in Chapter 3 and is illustrated Figure 6.3. An atomic component is one that does not contain further structural definition and therefore cannot be further subdivided. A composite component on the other hand contains atomic types or other composite types so it can be further subdivided. In the case that an included component is abstract, the algorithm identifies specializations of that component which are not.

In the example model fragment, the potential components include the *OCPowerUnit* and *OCValveBlock*. These components are not atomic types, instead they each contain further definition. For instance, the *OCPowerUnit* includes a pump, tank, and relief valve. The internal structure is illustrated in Figure 6.5.

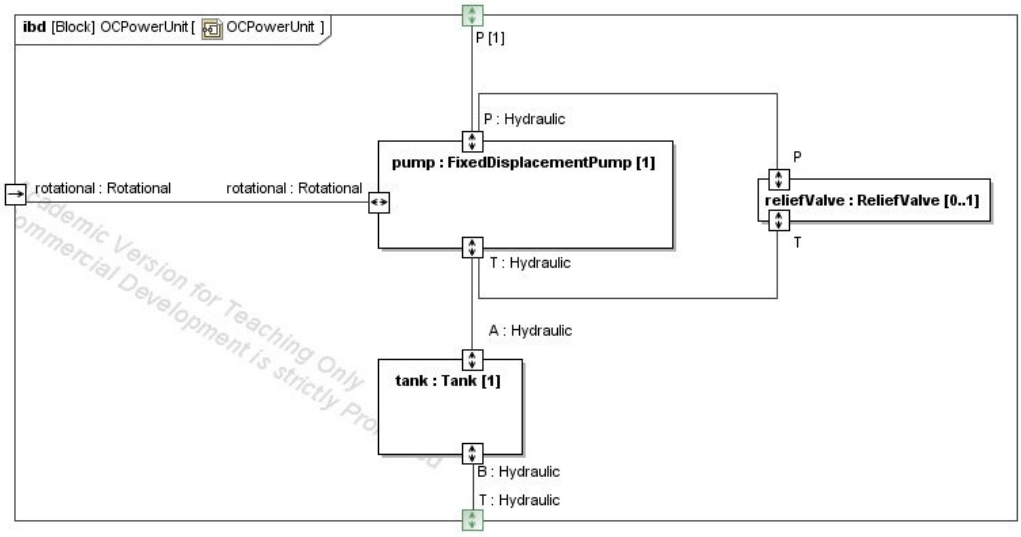

Figure 6.5: The internal definition of the *OCPowerUnit* functional unit.

These components are not contained in the superstructure definition, instead only the atomic types that make up these components (the pump, tank, and relief valve) are included. The choice of atomic versus non-atomic components is a modeling decision that is affected by the problem being modeled and should be chosen by the designer or modeler. In this example, if the abstract non-atomic component *ValveBlock* is included instead of *OCValveBlock*, then the algorithm would continue after identifying both *OCValveBlock* and *CCValveBlock*.

Each non-abstract atomic component type that is identified is re-declared as a new type that is referenced by the superstructure. The non-abstract components are re-declared separately from the problem definition.During this re-declaration, any elements in the component that are inherited from other component are included in the resulting flattened type definition. In addition, any equations related to the original type that have been defined in an analysis library are also included. The actual variables included in this re-declaration are copied from the analysis library. To identify the appropriate analysis

library model elements for each type, the transformation identifies links between structural and analysis elements that have been encoded in SysML AssociationBlocks. These association definitions represent re-usable templates that describe how structural elements relate to analysis elements. The definition of these AssociationBlocks was described in Section 4.3.4.

Once all the necessary component types for the superstructure have been defined, the next step is to create all the appropriate usages in the superstructure. Creating the component usages is a more complex process than creating the appropriate types because the number of usages is important and also they must be appropriate tracked so that connections can be added in the next stage. Also, each single usage in the original definition maps to some number of usages in the superstructure depending on the multiplicity of the usage. Also only atomic usages are included (usages that are typed by atomic components). For composite components, several usages are added. For each usage of an abstract type, multiple usages are also added to the superstructure; these usages are typed to the concrete types that specialize this abstract type. The flattened usages for this example are shown in Figure 6.6. The usages of the tank, relief valve, and fixed displacement pump are flattened from the *OCPowerUnit* and the check valve and directional valve are from the *OCValveBlock*.

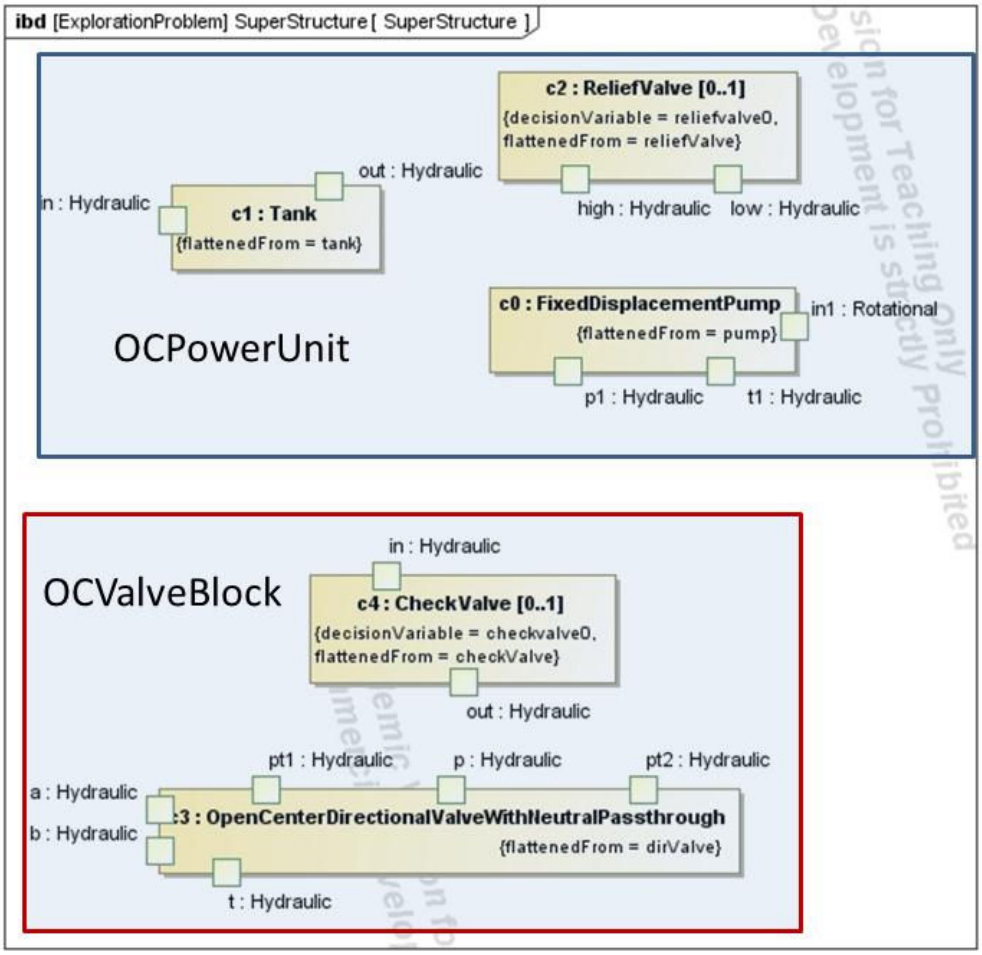

Figure 6.6: Components resulting from flattening of the original space definition

Once all usages are present in the superstructure, the next step is to add the corresponding connections. Some of these connections are always present, as described earlier in Section 3.4.2. These connections are copied into the superstructure by tracing the correspondence between usages in the superstructure and original definition as in the transformation presented in Chapter 4. In addition to these mandatory connections, there are optional connections which are also specified in connection templates captured using AssociationClasses as described in Chapter 3, Section 3.4.3. Again, these AssociationClasses can be defined between abstract or composite types so again a

resolution process is needed to insure that all the appropriate optional connectors appear in the superstructure. For AssociationClasses between composite types, the appropriate port needs to be identified in the flattened structure by tracing the connection between the interface of the composite type and the interface on the constituting component. Then, the links between this constituting component in the definition and the possible multiple instances that result in the superstructure are then traced to identify which ports should be connected. A similar process is used for abstract types where the flattened usages resulting from the abstract type are traced and connectors are instantiated. The flattened view with connectors for the running example is shown Figure 6.7. Some of the connectors are always present and simply directly copied, for instance the connectors between the relief valve and fixed displacement pump which are based on the structure of the *OCPowerUnit*. Others, stereotyped with «*OptionalConnector*» (this stereotype was defined in the profile shown on Figure 3.6 (on pg. 78), are included because of relevant connection templates. For instance, the connections running from the pump to the valve are part of connection template 3 from Appendix A.

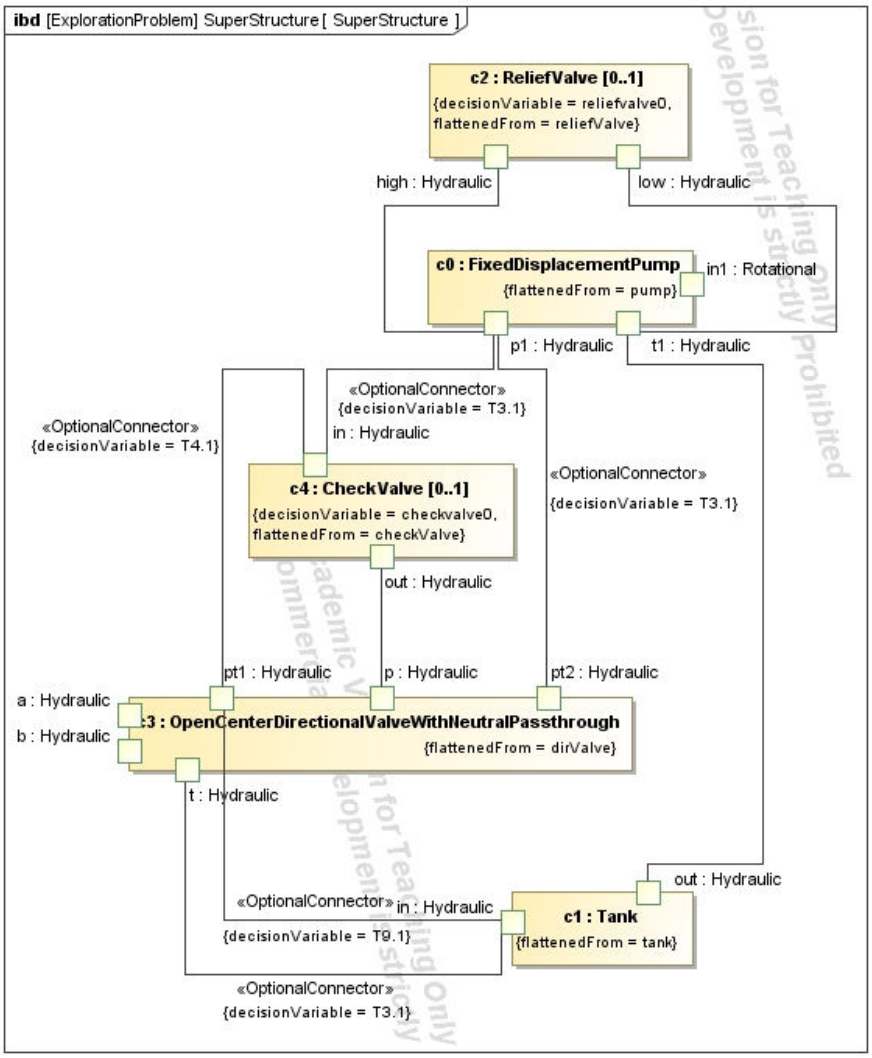

Figure 6.7: Flattened view with corresponding connectors.

Once the superstructure is constructed, it contains all possible components and connections. The next step is to include variables that capture whether particular components and connections exist within a particular alternative. These variables, labeled decision variables in the previous chapter, are added to the superstructure during the transformation process by identifying areas in the problem definition where there is potential for an alternative's structure to vary. These variables, which were more thoroughly described in the previous chapter, are included in the generated code and

values for them are selected by the solver during the search process. Whereas in the previous chapter the choice of these variables is considered known, here the variables are only implicitly encoded in the structure of the design problem and need to be explicitly defined. The selection of these variables is important because they must accurately describe the design space as captured by the encoded architecture selection decision. On the other hand, the more variables that are included in the formulation, the more difficult it is to search the space.

The first step is to identify the decision variables related to optional connectors. Groups of connectors are assigned a single decision variable based on the connection templates. For instance, in Figure 6.7, most of the connections running from the pump to the valve are flattened from the *T3* connection template and therefore have the same decision variable. If multiple pumps or valves were included in this model fragment example, connections running from each pump to each valve would be assigned to different decision variables.

In addition to the optional connectors, there is also other structural variability in the model. For instance, when usages typed to abstract types in the model are replaced with multiple concrete usages and types, and only one usage is present in any architecture. Therefore, additional decision variables are needed to capture this fact. Also, throughout the definition of the architecture, there are optional components that are included within the Functional Units (as described in Section 4.3.1). For each of these optional components, a decision variable is added.

The next step is to incorporate the knowledge captured in the tests into the superstructure. As discussed in Section 3.4.1, there are two types of tests. The first type

of test simply requires some composition of component attributes into a single system-level attribute. In this case, an additional variable is added to the superstructure to represent the system-level attribute. In addition, a constraint is added which is owned by the superstructure which describes which component-level attributes should be composed. One example of such a test is a mass test where the composition involves adding up the mass attribute of each component into a single system-level mass.

Including knowledge from the other type of test is more complex. Here, both the test structure and test procedure need to be incorporated into the superstructure. The first step is to include the test structure as a part of the superstructure. Then, the test procedure is converted into a set of constraints and this set is added to the superstructure.

To add the test structure to the superstructure, the same process that adds potential components to the superstructure is used. For each type of test component in the test structure, a new local re-declaration is created. The model properties and constraints from the relevant analysis models are copied into this new model as described previously. Then, for each usage of the component in the test structure, a corresponding component usage is created in the superstructure and typed to the new re-declaration. Then, connectors are created for any connections that exist in the test structure between test components or between the test components and the interfaces of the system boundary. For connections between test components, the connector connects the appropriate interfaces. Because the interfaces of the system boundary do not exist in the superstructure, the connections between the test components and the superstructure's interfaces become connectors between the appropriate test component interface and the appropriate interface of a potential system component. In order to identify the appropriate

interface, all connections and potential connections that exist between the system boundary interface and component interfaces within the system are traversed. If the interface is on an atomic component which has no further internal structure, the corresponding interface in the superstructure is used. If the interface is on a functional unit or some other composition of components, the process continues until it identifies an atomic component.

One important consideration, as mentioned in Section 3.4.1, is that multiple tests may reuse the same structure. Therefore, it is important to identify which structural elements have already been included in the superstructure so as not to re-include these elements. This is true for both the test components and also the connections.

As is apparent from the transformation process, there are a large number of potential links between elements in the superstructure and the original problem definition. These potential links are important not only for documentation purposes but also to help identify certain elements in the superstructure during the transformation process. To capture these links, certain superstructure elements are stereotyped using the «*FlattenedComponent*» or «*FlattenedValueProperty*» stereotypes (also defined in Figure 3.6, pg. 78). These elements can then use the flattenedFrom tag value to refer to the original component or value property. Another possibility for modeling these correspondences would be to simply use SysML connectors between elements. Stereotypes were chosen in this case because visually it was easy to identify which superstructure elements corresponded to which problem definition elements by just looking at the superstructure elements. In the connectors' case, the user would need to refer to a separate diagram which included the connectors.

6.4 Generating AIMMS Code

Once the superstructure is created, the second part of the transformation process is to generate executable AIMMS code. AIMMS has a textual representation language with a particular syntax which must be followed when generating this code.

Also, when the generated code is imported into the AIMMS tool, there are some practical considerations which must be addressed. For instance, the importer does not allow lines of more than 255 characters, which restricts the length of variable names and also requires that large constraints or data sets must be broken over multiple lines.

In the superstructure, the individual constraints are not manipulated and instead are just copied in directly. It is during the pretty printing process where textual manipulation of the constraints occurs so they conform to the expectations of the AIMMS interpreter.

There are a number of model-to-text transformation tools that can simplify the definition of this transformation process. For practical reasons such as the need to adjust the output text based on the limitations of the AIMMS importer mentioned previously and also the additional training needed to use these tools, instead of using any of these tools the pretty printer was written manually.

Even though the superstructure representation has more of the hierarchical structure of the original representation removed, there is still some flattening that occurs in this step. In the SysML model, there is a type-usage relationship wherein a particular variable or constraint is only defined once. In the AIMMS language, this must be completely flattened and the variable or constraint must be defined each time.

The pretty printing process has the following steps, based on the mathematical programming representation described in the previous chapter:

1. The number of each type of variable (variables with the same name and type) in the superstructure is counted and an AIMMS set is printed for each (Although this could also be grouped by components).
2. AIMMS variables are printed corresponding to the variables. They are indexed by the appropriate set.
3. Component constraints for each component usage that is present in the component type are printed as AIMMS constraints. Before printing, variable names in the constraints are replaced with the appropriate name and index based on step 2.
4. Constraints for the connections (and optional connections) are printed as AIMMS constraints.
5. Constraints for the tests are printed as AIMMS constraints.
6. The object and other parameters are printed in the AIMMS mathematical program construct.

6.5 Import-Export SysML

As an aside, the SysML representation relies heavily on model libraries which contain reusable domain knowledge that are independent of the problem definition and can be leveraged for different design problems. Once these model libraries are defined and available to a designer, the cost of creating the problem definition is greatly reduced. But, defining these model libraries is not a trivial task. One approach would be to import the knowledge into SysML from existing sources.

It is always important to consider how knowledge represented in a legacy format or in existing models can be imported into SysML. Also, simply asking the engineer to represent all of his knowledge within SysML manually has a number of inherent disadvantages. Many times, domain-specific tools offer a more effective user interface and a more natural presentation of the engineer's knowledge. Also, many times the engineer has significant training and experience with a particular domain-specific tool making the tool much easier to use. Transitioning to SysML would require significant re-training of the engineer which would come at significant cost. A better approach would be to allow the engineer to represent his knowledge in his native form and then import that knowledge into SysML.

Most current transformation approaches are focused on transforming a SysML model into some other representation that is capable of analyzing the model. An additional issue is maintaining the consistency between the SysML model and the resulting analyses.

6.6 Summary

In this chapter, a two-stage transformation approach was presented for transformation from the SysML presentation of an architecture selection decision into a mathematical programming optimization problem. A mathematical programming solver can then operate on this formulation to perform an early stage architecture exploration and identify feasible or promising solutions as will be demonstrated in the following two chapters.

This transformation provides some tangential support for hypothesis 3; one of the current drawbacks with using a mathematical programming based approach to perform

architecture exploration is the difficulty of creating the problems; without the ability to conveniently generate these formulations the approach is not practical. A transformation approach, such as the one presented, greatly reduces the difficulty of generating these problems making the use of a mathematical programming based approach more practical.

CHAPTER 7:

EXCAVATOR EXAMPLE

In this chapter, the excavator example is presented to demonstrate the applicability of the proposed approach to real-world scale problems. The excavator example relates to the selection of an architecture for the hydraulic subsystem of an excavator. An excavator is a piece of off-road construction equipment that is expected to have the flexibility to perform a number of common tasks, from relocating soil for the purpose of digging trenches or preparing a landscape for commercial development to lifting heavy objects. The hydraulic subsystem actuates a four degree-of-freedom mechanical system that performs each of these tasks. This mechanical system is connected to four cylinders which actuate the vehicle's arm and a hydraulic motor which rotates the structure connected to the undercarriage. An illustration of an excavator is shown in Figure 7.1; this illustration is taken from (Haga, 2001).

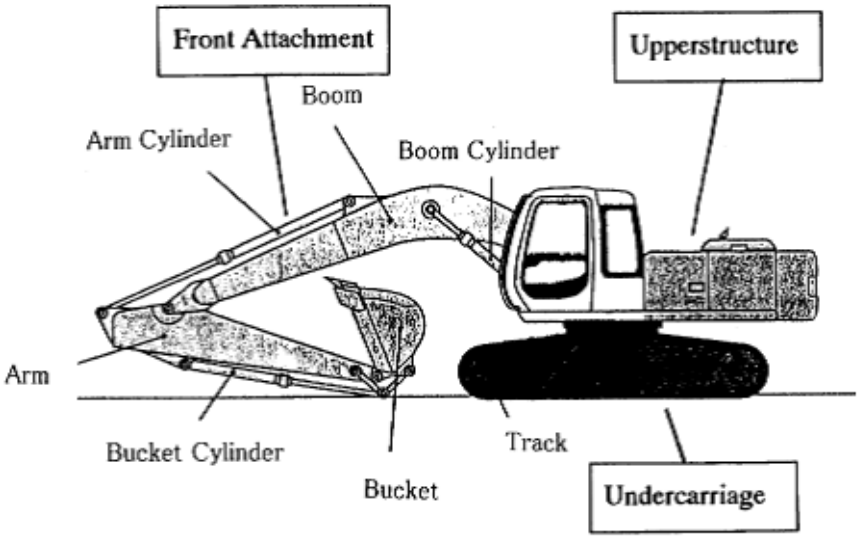

Figure 7.1: Excavator, taken from (Haga, 2001)

The excavator is chosen as an example because of the potential variability in the hydraulic subsystem. There are a number of candidate architecture configurations, with potential variability in the number of pumps, valves, and prime movers (engines) included in the architecture. Although most configurations have the same number of valves, multiple functions can be combined differently (for instance, in most excavator architectures a single valve actuates both boom cylinders because they are kinematically linked)

Here, the focus in on purely open-centered architectures, although traditionally the architectures can be either open-centered or closed-centered. In an open-center architecture, the neutral position for the directional control valves allows fluid to pass through the valve. In such an architecture, the pump can continuously produce flow, so fixed-displacement (or constant-displacement) pumps can be used. These pumps are usually cheaper and simpler than variable-displacement pumps. When all of the valves are in the neutral position, flow from the pump bypasses the valves and goes back to the tank. In a closed-centered architecture, the neutral position of the valve does not allow fluid to flow through the valve (the neutral position is closed). In this case, a variable-displacement pump is needed because in the neutral position there is no path for further flow. Closed-center architectures are more energy efficient than open-centered architectures although they are also more complex and have higher component cost. Because the goal of this section is to provide a proof of concept, only the simpler open-centered architectures are considered. One objective is to minimize the fuel consumption while the hydraulic subsystem actuates the cylinders. Fuel consumption is chosen as the objective because current industry trends and government regulations are pushing for

more economic vehicles. The other objectives considered are to minimize component cost (facsimile for manufacturing cost) and a life-time cost (component cost plus fuel cost over the expected lifetime of a vehicle).

The use scenarios for the hydraulic subsystem are also considerably simplified. As mentioned earlier an excavator is often utilized in a number of scenarios, including approximations of each of these scenarios as part of the architecture exploration problem would make it very difficult solve. Instead only five use phases are considered; these use phases are fairly generic but cover of the scenarios an excavator would perform generically. These five phases are: one where no cylinders move, three where each cylinder (degree of freedom, both boom cylinders are always actuated together) is actuated separately (the swing is neglected), and one where all four cylinders (three degrees of freedom) are actuated together. This insures that any candidate architecture that is considered feasible by the solver is able to perform the rudimentary tasks needed for the excavator to function. More comprehensive analysis of the system's performance can then be performed in future steps.

The goal of this chapter is to demonstrate that the approach is applicable to real world problems in order to test and support hypothesis 1 and hypotheses 3:

H1. Designers can represent their architecture exploration problem in information models using a domain-specific language consistent with decision theory.

H3. Designers could use mathematical programming techniques to identify promising solutions early in the exploration. Mixed-Integer Linear Programming should be used for architecture selection.

Because common architectures of the hydraulic excavator are well known in industry, the design of the excavator provides an excellent case study for the method. Since an extensive array of knowledge about potential system architectures for this system exists, it is easier to quickly identify whether the solution process is indeed identifying feasible solutions. The more difficult problem is to ensure that the resulting solutions truly span the potential design space. Although significant prior exploration into the design of excavators does provide some guidance about feasible and promising architectures, without conducting an exhaustive search, it is impossible to determine if the solution approach has truly identified all possible promising candidates. The other consideration is what makes a particular solution candidate a promising candidate: is it simply that a particular candidate solution is able to perform all of the prescribed use scenarios or should the solution also meet a certain threshold for the objective value? In this investigation, a candidate solution that is capable of performing the prescribed use scenarios is considered to be promising.

To better understand how this chapter fits in with the rest of the thesis, the goal is to build on the work presented in Chapters 3-6. The underlying approach to formulating the problem in SysML along with the necessary modeling constructs was presented in Chapter 3. To generate the mathematical programming problems, the code used to implement the approach in Chapter 6 is used. Unlike Chapter 5 where the mathematical programming formulation is presented in a generic form, in this chapter there is additional focus on providing a more concrete example of the process and highlighting the issues that arise during this particular problem formulation. How these issues are

resolved can be used as a generic starting point or a set of best practices for other problems.

The rest of the chapter is outlined as follows. The definition of the excavator example using the SysML representation from Chapters 3 and 4 is presented in the next section. An outline of the considered optimization problems and how the excavator problem is represented as a mathematical programming optimization is considered in Sections 7.2 and 7.3. Some simplified versions of this problem definition are used as verification examples in Section 7.4. The optimizations for the full version of the problem are presented in Section 7.5. Then, two potential approaches for mitigating scaling issues are demonstrated in Sections 7.6 (where the problem is further constrained) and 7.7 (where system sizing is neglected). Based on the results of this chapter, the overall approach is compared to two other related approaches in Section 7.8.

7.1 Defining the example in SysML

To help illustrate the language presented in Chapter 3, it will be used to define the architecture selection decision being considered in this chapter. In the design of a hydraulic excavator, the mechanical subsystem is rarely changed because of the high cost necessary to change the manufacturing process for the mechanical structure. Also, the excavator's hydraulic subsystem has much greater opportunity for variability in terms of the desired architecture, whereas in the mechanical structure only the geometry of the components would change. Taking this into consideration, in the problem definition presented here the definition of the mechanical subsystem is considered to be fixed and the definition of the hydraulic subsystem is left unspecified.

To begin the definition of the architecture exploration problem, it is important to capture the requirements and selection criteria for the candidate architectures. Based on the evaluation criteria, the designer can then define an appropriate space of candidate solutions that will be explored and the appropriate evaluation criteria for each of these solutions.

There are a number of different ways to define selection criterion, but the criterion chosen must be able to rank-order each of the different architectures. Part of the difficulty in evaluating the architectures is their changing structure; to not bias the exploration process, the selection criterion must be sufficiently independent of a particular architecture alternative. For instance, it would not make sense to compare an all-electric vehicle to a traditional vehicle using a metric such as gasoline consumption. When trying to minimize the fuel consumption, any electric vehicle would be evaluated to be superior even though it may use require significantly more energy and from a broader perspective would be the poorer choice.

A sample requirements diagram for the actuation subsystem is shown in Figure 7.2. In this requirements diagram, the requirements on the hydraulic subsystem are broken down by performance, cost, and mass. In the example mathematical programming formulations, the mass requirement is neglected because cost and mass are so strongly correlated (in this domain).

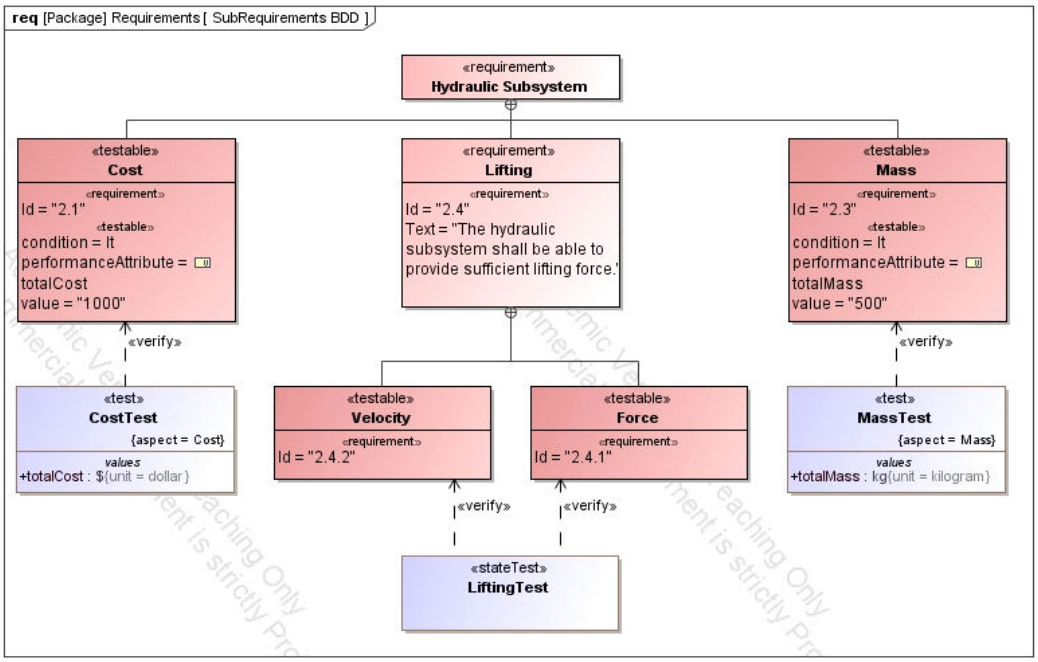

Figure 7.2: A Requirement Diagram for the Hydraulic System that also includes the proposed testable requirements

Once the requirements are defined, the next step is to define the various related test cases. In the previous diagram, three test cases are considered, a lifting test, a mass test, and a cost test. The lifting test is defined through the use a test context, because it is likely that a real-world specification would include multiple different test cases for dynamic behavior, although each of these test cases would be based on the same structure. A test context for the excavator subsystem is shown in Figure 7.3. Here, the subsystem is connected to loads, which can be specified by the test procedure. Also, a fuel tank is connected to the subsystem to allow the fuel consumption to be measured.

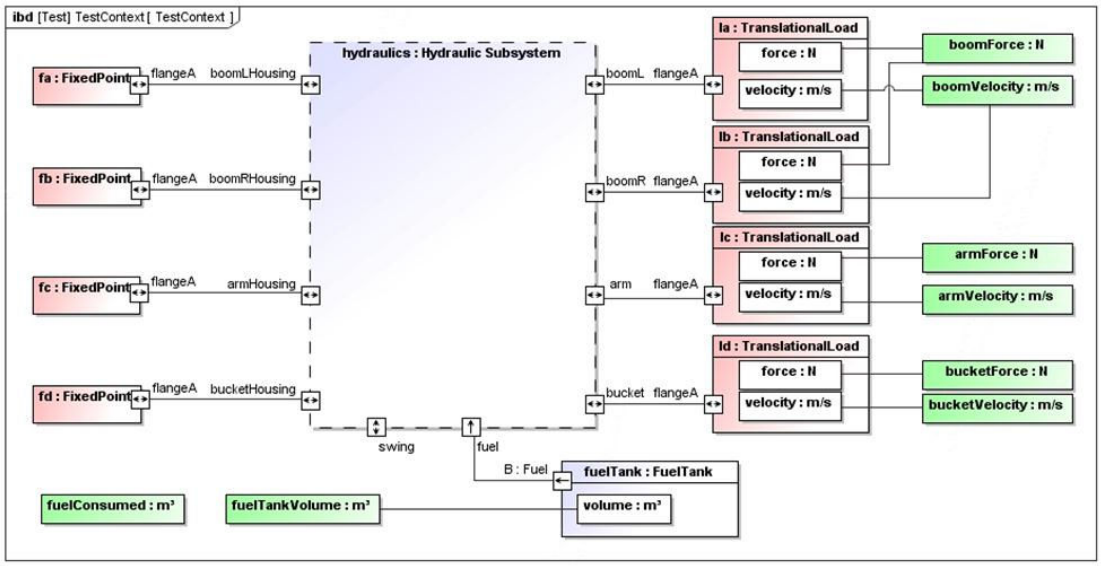

Figure 7.3: Test context for excavator subsystem including the IBD for the hydraulic subsystem.

To define the test procedure, a SysML activity is created which encompasses the entire procedure. SysML actually supports multiple different formalisms which would be suitable for defining the test's behavior, but the activity formalism was chosen because it seems to be the most intuitive to designers and also because there are constructs to relate the activity elements to the test structure.

The test cycle for the lifting test is shown in Figure 7.4. This test cycle can be broken down into 5 distinct operating phases, as is visualized in Figure 7.5. There is a stage where none of the cylinders move, a stage where each cylinder moves independently, and finally a stage where all of the cylinders more together.

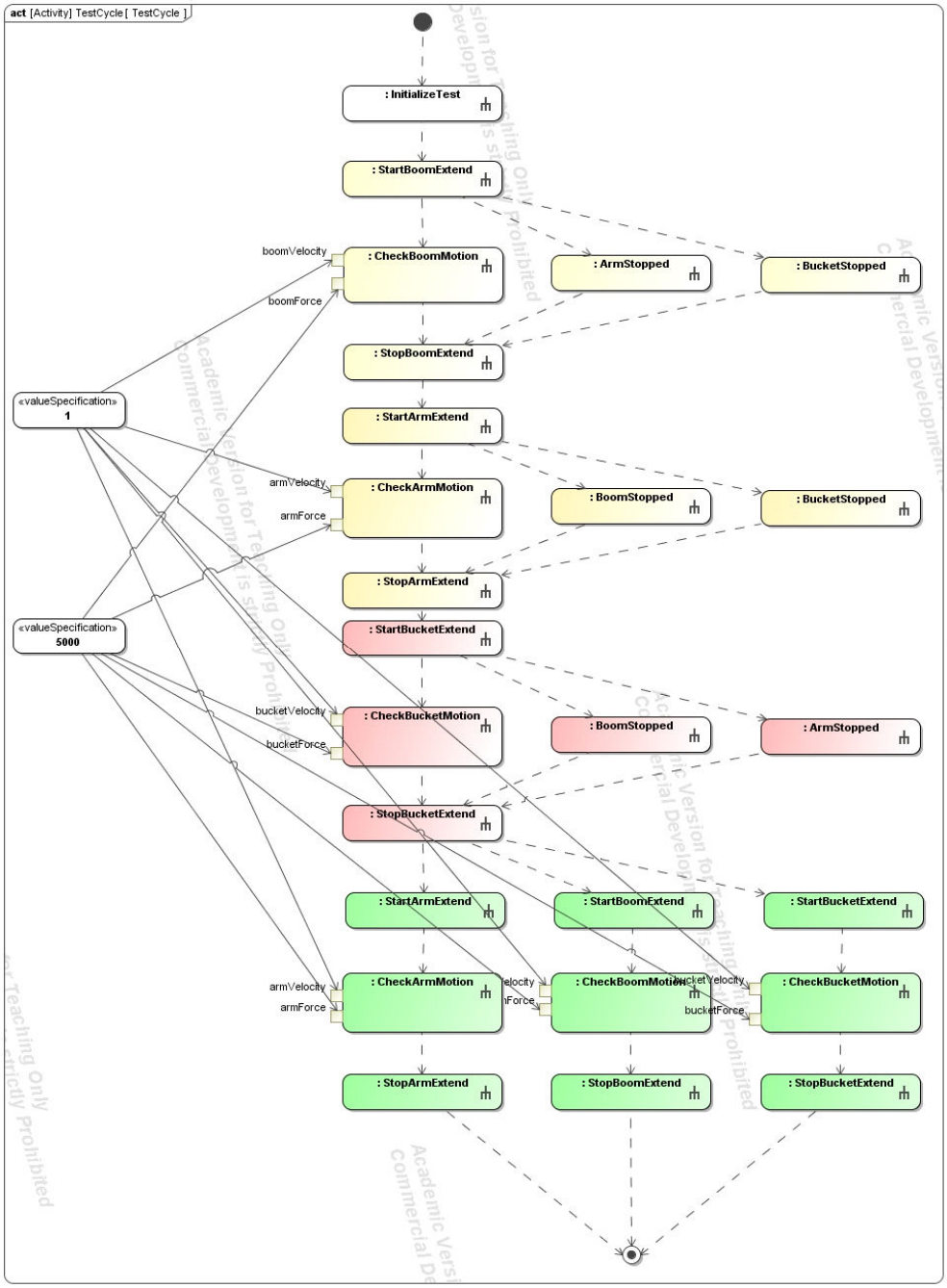

Figure 7.4: Test cycle for the actuation subsystem

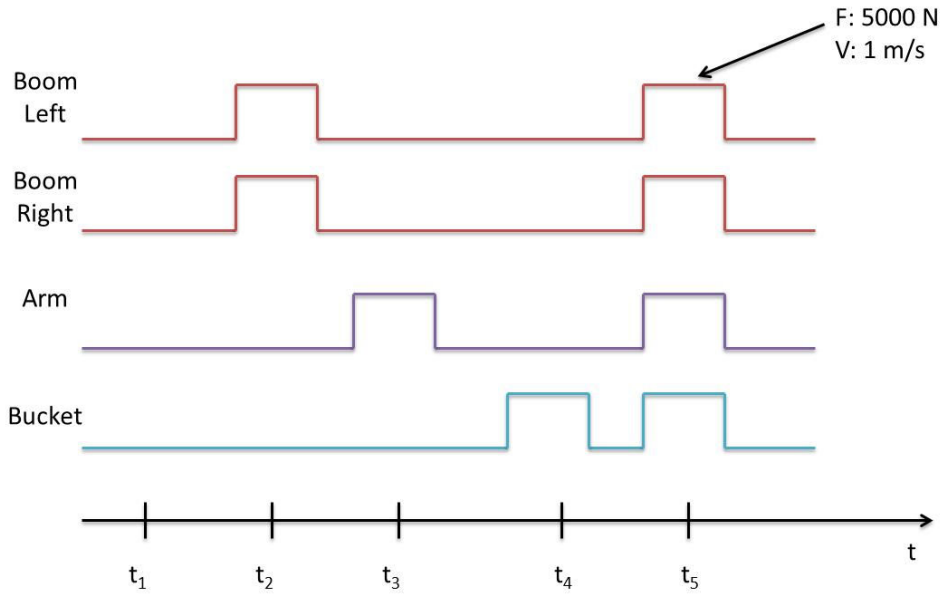

Figure 7.5: Visualization of the test cycle for each cylinder.

Once the designer has created the requirements and selection criteria, the next step is to define the space of potential solutions. When defining the space of architectures, the designer needs to capture two basic facets: all of the potential components that can be used as part of the architecture and all possible connections between these components. The potential connections have been defined by the connection templates found in Appendix A.

In this example, the excavator has 4 cylinders, which are used to actuate the different parts of the arm, and then a swing motor to swing the cab. These are fixed parts of the hydraulic subsystem which are always included. In addition, the subsystem has some number of prime movers (engines), pumps, and directional valves.

The potential components that can be included are shown in Figure 7.6. The hydraulic subsystem is composed mostly of functional units, which reflect common combinations of components. In the hydraulics domain, pumps are often connected to

hydraulic tanks which store the hydraulic fluid. By combining these into a single "Power" functional unit, the designer does not need to include both the pump and tank in the configuration each time. In addition, the common configuration between the pump and tank can be applied in each potential architecture configuration; during the solution process this configuration does not need to be rediscovered. The pump type has properties and interfaces that would be common between different pumps categorized by this type. It does not have any values that clearly represent a single pump instance, for example no knowledge is captured about the size of the pump or a particular pump brand. The internal specification of the hydraulic subsystem is shown in Figure 7.7. The connections between the cylinder interfaces and the interfaces of the hydraulic subsystem are fixed. The rest of the structure is left undefined. The multiplicities on the part properties represent the number of each component.

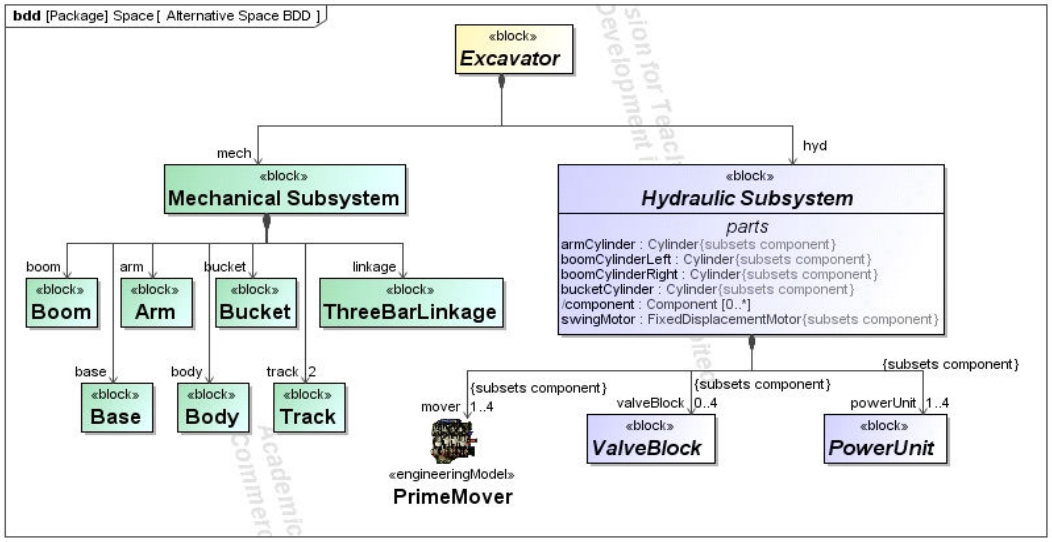

Figure 7.6: Block definition diagram representing the entire excavator structure.

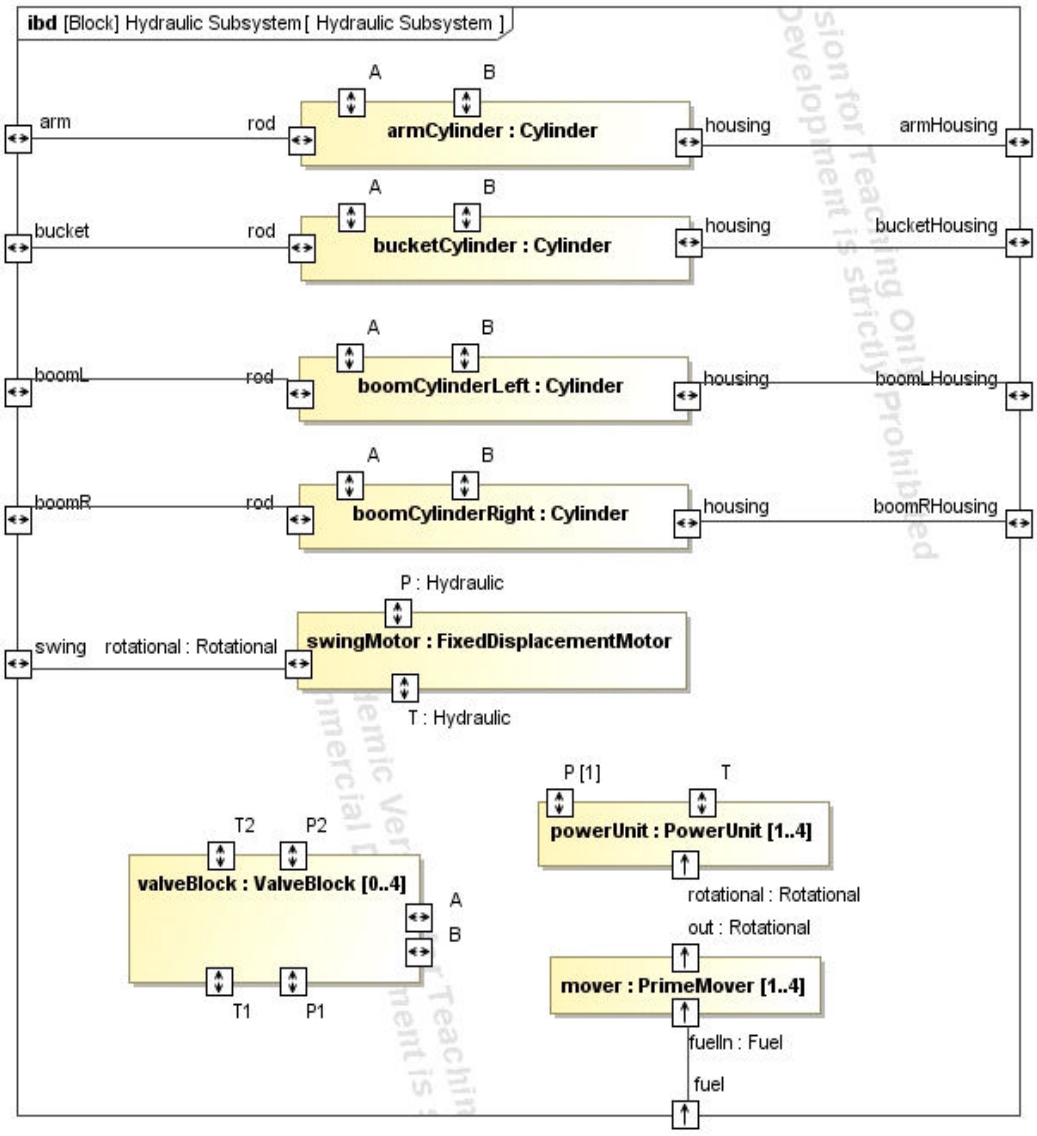

Figure 7.7: SysML Internal Block Diagram of the excavator showing the partially specified hydraulics subsystem.

The set of potential components and connections defines the space of candidate architectures. Although in this form the space is very difficult to visualize, it does provide a very compact representation for a large number of potential solutions.

7.1.1 Comparison to other architecture exploration problems

In this investigation, the example problems come from the hydraulics domain. It is important to consider how this would approach would apply to other domains. It is also important to consider how the problem definition changes for other examples.

For this example problem, the assumption is that model libraries are available. These model libraries include structural component libraries that include available hydraulic components, libraries that include available off-the-shelf components, algebraic analysis models, and the correspondences between the structural and analysis models. On the other hand, the statement of requirements and related tests as well as the definition for the space of solutions is unique to the excavator problem. Although significant investment is needed to create the hydraulic domain libraries, within an institution that is designing hydraulic equipment these domain libraries could be used on other hydraulics related projects.

The hydraulics domain has several interesting characteristics which make it a good fit for the approach used in this investigation. Another important consideration is how the approach would apply to other domains. In this investigation, one of the underlying assumptions to the approach was that an architecture exploration problem is a choice between systems which are defined as a composition of well-defined components. In the hydraulics domain, components are modular in nature, the types of components are well known, and hydraulic systems are often described as a composition these known components. This is not the case in every domain, but when considering current practice in systems engineering, it is common for systems to be represented as a composition of known components.

The major issue when applying this approach to other domains is the applicability of the composition process, in particular the composition of component-level analysis models into system-level analysis models. A number of analysis tools and languages (such as Modelica) rely on a similar composition process to reduce modeling effort, where model library fragments are composed into more complex analysis models. These previous approaches have demonstrated that such a composition approach works for physics-based behavior models in a number of domains, such as simulating the dynamic behavior of mechanical or electrical systems.

The other consideration is whether such an approach is useful for a particular problem or problem domain. When considering the overhead of explicitly modeling the architecture exploration problem, this approach is the most applicable for problems where designers are interested in exploring a large number of system architectures. In those cases, the upfront effort of explicitly modeling the problem is mitigated by the time savings of not manually creating the related analysis models. If the number of architecture configurations that a designer wants to consider is small, the initial investment of time may not be mitigated by future time savings. That being said, even for such problems there is value in explicitly modeling the problem.

To apply this approach in other domains, the first step would be to create the relevant model libraries. Although this is described as an upfront process, in practical applications it is likely that this would be an iterative process where components would be added as needed during the definition of the problem. To create the model libraries (and other models that include domain-specific knowledge) the relevant domain experts would need to be engaged. In conjunction with the definition of the model libraries,

systems engineers would define the objectives and requirements for the system. Test engineers would then create the architecture independent tests. Then, systems engineers in conjunction with the domain experts would define the space of possible solutions. This would complete the definition of the architecture exploration problem.

7.2 Outline of Optimizations

In this section, the plan for supporting H3 in this chapter is outlined. Before considering the full excavator example, several smaller examples will be considered to establish that the mathematical programming formulation is indeed applicable and that the solvers are capable of solving this type of problem. Also, the smaller examples allow for more comprehensive checking and verification of the code that results from the model transformations than the full excavator example. These smaller examples will also allow for a more comprehensive number of experiments to test the approach and understand how the inclusion of different knowledge from the problem definition affects solution time and solution quality. For the full-fledged excavator example experimentation is limited to demonstrate that for a single version of problem, the CPLEX solver is capable of finding feasible solutions in a reasonable amount of time and also capable of optimizing the problem with respect to cost and fuel consumption. A constrained version of the excavator example is also presented; this version is presented to support the argument that by constraining the problem one can reduce the optimization time while still maintaining much of the interesting space. Also, a version of the optimization is presented where sizing is not performed, demonstrating another potential avenue for managing scale.

The first step of in each section is to use the solution approach to generate a feasible candidate architecture. If the solution approach is incapable of this first step, further testing is not necessary. Also, verifying that a candidate architecture is feasible can be accomplished by comparing it to existing known configurations because in practice there are a range of well-known architectures for both the simpler examples and the hydraulic excavator. After it is shown that the approach is capable of generating a feasible architecture, the next step is to attempt optimization. Using the solution approach to generate feasible candidate solutions and optimized solutions is strong support for hypothesis 3.

When considering the size of this optimization problem and the speed in which solutions are generated and comparing it to current state-of-the-art approaches being employed for computational design synthesis, it appears that this search approach is better. The validity of this statement is discussed along with the supporting arguments with the full excavator example problem. Although this is only for this particular example problem, when comparing to real-world systems engineering problems, the size and scope of the examples presented here do provide an approximation of real world applications.

Another issue is that these examples stem from the hydraulic systems domain, and there is a lack of focus on designing the controller for the system and only a few components from other domains. Although controller design is neglected, optimal control often uses mathematical programming techniques to design controllers (Sager, 2012).

7.3 The Mathematical Programming Framework

Although Chapters 5 and 6 present a structured approach to generating a mathematical programming formulation from the related SysML description of an architecture selection decision, in this section some additional discussion is provided on the mathematical programming optimization problem specifically for the excavator example.

In the example problem, the selection is between a number of different configurations that include a variable number of pumps, valves and engines. The number of cylinders is a fixed set because the assumption is that these are the only option to actuate the system. The search space could be further extended by considering different cylinder types, for instance both single-acting and double-acting cylinders instead of simply double-acting cylinders.

The description of the problem leads to a number of binary variables: one set to describe potential connections between these components and one set to describe optional components. When connections are grouped together and only common connection types are included (pumps connected to valves, valves to cylinders, engines to pumps) there are 104 binary variables. These binary variables will be referred to as decision variables because the set represents the alternative choices.

For each potential component, there are a number of variables and constraints that are needed. To simplify the definition, the variables for the component interfaces are instantiated first. In this example, that includes the flow and pressure at every hydraulic port along with the force and velocity produced at the cylinders and the torque and

angular velocity out of the engines and into the pumps. Each of these variables is indexed by the system states; in each usage scenario the values of these variables will change.

For each component there are a number of other variables; these can be grouped loosely into two sets: those that describe the components sizings (its sizing parameters) and those that describe internal component behavior. For each cylinder, its sizing parameters include the stroke length, and the areas on the rod and bore side. Classically, these are represented as the rod and bore diameter, but that would involve a (simple) nonlinear constraint so instead they are represented by the areas.

In addition, there are variables that describe the internal component behavior. For the cylinder this may include the pressure differential across the piston or the force differential produced by the cylinder. Unlike the sizing parameters, these vary with the system states.

The algebraic constraints found in this section and throughout the algebraic library are derived from the Parker Hannefan Design Handbook (Parker, 2002) for hydraulic components and the McCandlish model for pumps and motors (McCandlish, 1984). Component sizes are based on the databases used by Shah in previous work (Shah, 2010c). A full listing of these can be found in Appendix A.

Now the algebraic constraints that describe the behavior need to be considered. As a reminder, each cylinder has two ports, labeled A and B. In this model, these constraints are steady-state equations based on first-principles. For the cylinder, the equations are:

$$\forall s \ (F_s = A_{\mathrm{b}} p_{\mathrm{A,s}} - A_{\mathrm{r}} p_{\mathrm{B,s}})$$

$$\forall s (A_{\mathrm{b}} Q_{\mathrm{A,s}} + A_{\mathrm{r}} Q_{\mathrm{B,s}} = 0)$$

$$\forall s (A_{\mathrm{b}} v_s = Q_{\mathrm{A,s}})$$

where s represents the different use scenarios, these equations are active for all use scenarios, F_s is the output force for each use scenario, v_s is the velocity of the cylinder for each use scenario, A_b and A_p are the bore-side and rod-side area respectively (note, the rod-side area is the effective area that the fluid pushes against on the rod side, not the area of the rod), p_{A} and p_{B} represent the pressures at ports A and B, and Q_{A} and Q_{B} the follows through ports A and B. As presented, these equations are nonlinear because both the potential areas and pressure and flows and velocity are unknown variables and are used in products. There are several ways to address this issue; for instance, these products can be approximated using the interpellants described previously. In this problem, a different approximation technique is used that relies on the fact that A_{b} and A_{r} are sizing parameters and there are a limited number of discrete choices for them (based on the available components considered in Appendix A). For each potential combination of A_{b} and A_{r}, an optional constraint (an indicator constraint) is added with a given value for these two variables removing the product of two variables. Part of the consideration is scaling, usually the areas are very small quantities while the pressures are large. A set of binary variables are added to describe which of the potential combinations are selected.

The same can be done for any sizing situation where known components are being considered.

For the engine, the equations are slightly more complex. Since fuel consumption is an important characteristic of the system, it needs to be approximated. Also, the engine must produce torque that is constrained by the torque curve. The equations for the engine are:

$$\forall s\left(\mathrm{T_s} \leq f(\omega_s)\right)$$

$$\forall s(\omega_{\min} \leq \omega_s \leq \omega_{\max})$$

$$\forall s(\mathrm{T}_{min} \leq \mathrm{T_s} \leq \mathrm{T_{max}})$$

$$\forall s(P_s = \mathrm{T_s} \cdot \omega_s)$$

$$\forall s\left(r_s = f(\omega_s)\right)$$

$$\forall s(fuel_s = r_s \cdot P_s)$$

$$fuel_{\mathrm{tot}} = \sum_s fuel_s$$

where T is the torque produced by the engine, ω is the angular velocity, P is the power, r represents the fuel consumption per power at a given angular velocity, and $fuel$ is the fuel consumption per time and $fuel_{tot}$ is the total fuel consumed. The units of total fuel uses depends on the units of r, in this case r is selected to be approximately (depending on angular speed) 0.5 kg/W (r is based on common brake specific fuel consumptions for engines).

The equations for T and r are rewritten using the 1-D interpellant (described in Section 5.4.3) along with known values based on existing engines. The products can be

rewritten using the 2-D product interpellant (described in Section 5.4.3). As mentioned earlier, a full set of equations can be found in Appendix A.

7.4 Verification Examples

To generate the simplified examples, subsets of the excavator problem were taken. Instead of including all 4 cylinders and so forth, the first example (Labeled E1) includes only a single type of each component.

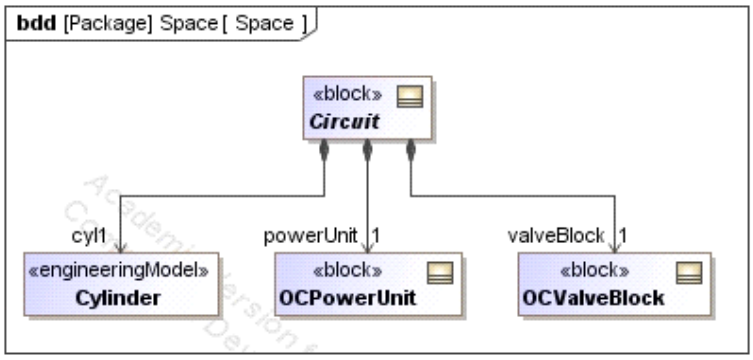

Figure 7.8: Structure for the verification example. Only one type of each component is included.

Although this is a simple example with only one feasible configuration and only 32 possible configurations, most of them not unique, it provides an excellent verification example for both the transformation process and the constraints used to model the components. The resulting mathematical programming problem is small enough that each transformation output can be manually checked. Also since there is only one feasible architecture configuration, it is easy to check that the solver is finding the appropriate configuration. If the solver finds a different (actually infeasible) configuration or if it finds that there is no feasible solution, then this is clear indicator that the constraints are wrong. In more complex problems, it is possible that the solver is simply incapable of

finding a feasible solution in a reasonable amount of time. The mathematical programming problem generated from this definition is still relatively large, approximately 2000 lines of AIMMS code. This formulation is shown in Appendix B.

It takes CPLEX 0.28 seconds to a find a feasible solution. This found solution is shown in Figure 7.9. This is indeed the only feasible configuration. In the illustration of the solution, the valve blocks and power unit have been flattened into their atomic components. In this configuration, the prime mover (engine) provides power to the pump. This pump provides hydraulic flow to the valve, which modulates the flow. When the cylinder needs to move, the valve moves to the on position (represented by an *on* binary variable in the problem formulation) and flow (and pressure) are provided to move the cylinder. When this is not the case, the cylinder is in the off position (presented by an *off* binary variable) where flow is allowed to pass back to tank. If the cylinder needs to move in reverse, the valve can be switched to a reverse position (represented by a *back* binary variable).

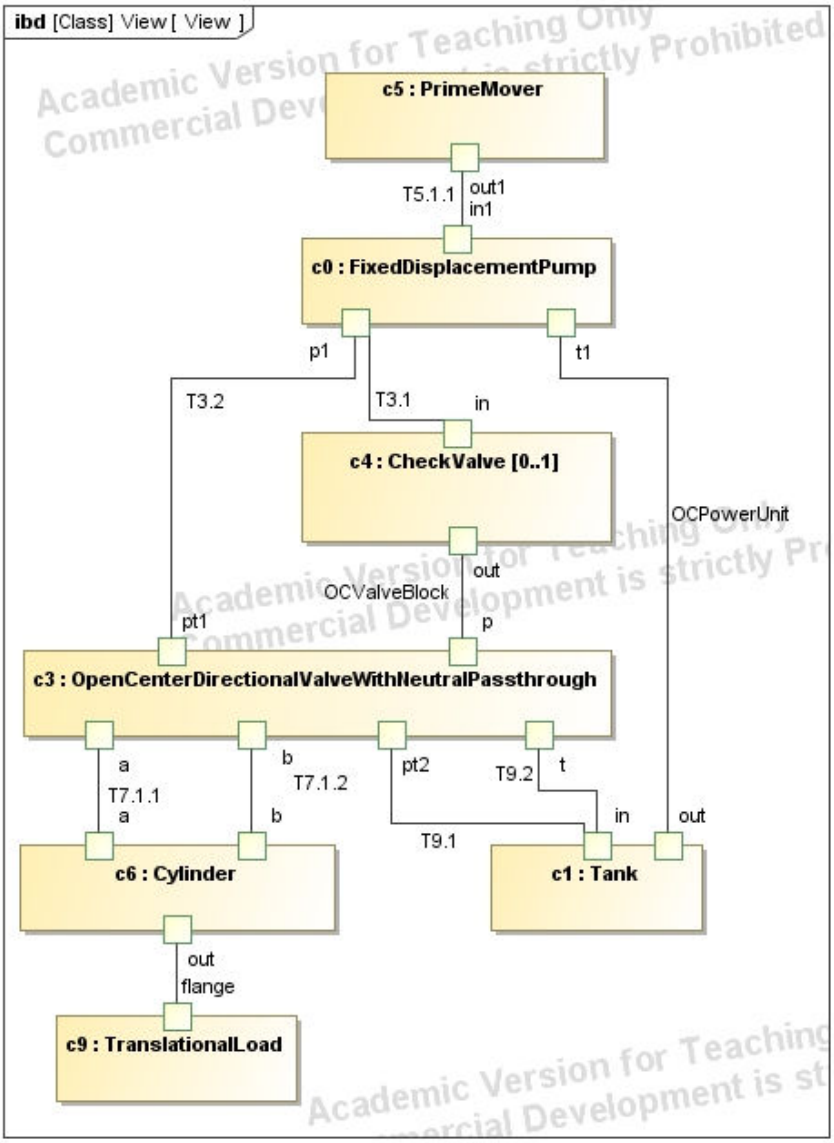

Figure 7.9: Resulting architecture from simple verification example.

The second verification example is more complex, including two of each type of component. This allows for a slightly larger number of feasible configurations, 3 unique in total, and more variability specifically in the number of pumps and engines used in the architecture. Unlike the previous problem, the search space is significantly large, with 2^{20} configurations (although not all of these configurations are unique). An illustration of this problem is shown in Figure 7.10 and Figure 7.11.

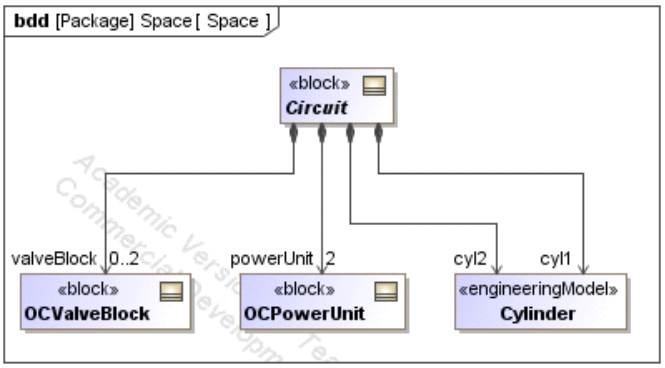

Figure 7.10: Verification example with two of each component.

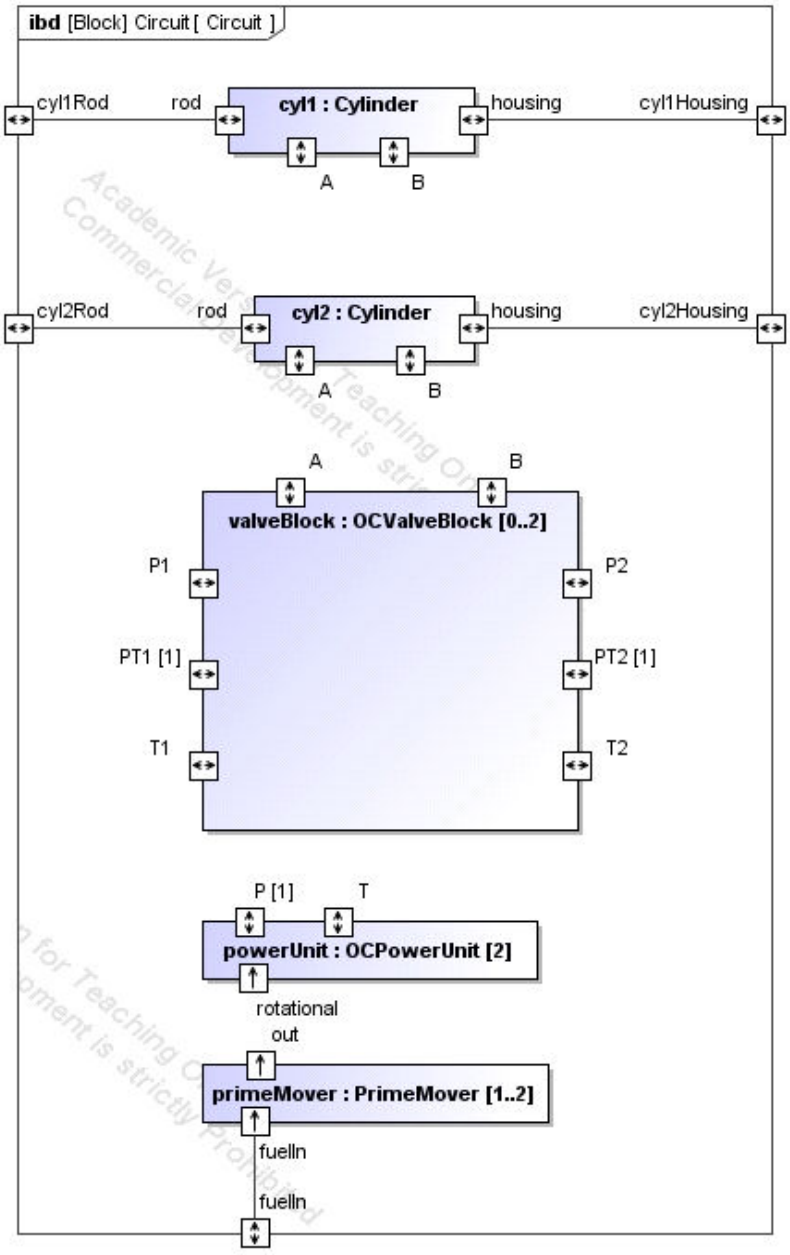

Figure 7.11: Internal structure of the second verification example.

With this verification example, the goal is to insure that the solver can find architectures that include both a single pump powering each valve and also architectures

where the valves are connected in series. Unlike the previous example, an optimization is run to minimize the fuel consumption where the objective function is:

$$fuel_{total} = \sum_{s} \sum_{engines} fuel_s \cdot t$$

where s is the number of scenarios, $engines$ is the set of engines, t is the time of each scenario (the assumption is each scenario takes an equal amount of time so this can be neglected or chosen to be an appropriate constant), and $fuel$ is the amount of fuel used by the particular engine during that particular scenario ($fuel$ is defined in the engine model described in the previous section). Since the goal is not to optimize fuel consumption, but instead to insure that the solver can indeed span the space of solutions, the solver time is limited in each example to 30 seconds. The first solution found is shown in Figure 7.12. In this example, there is one prime mover and one pump with the valves connected in series.

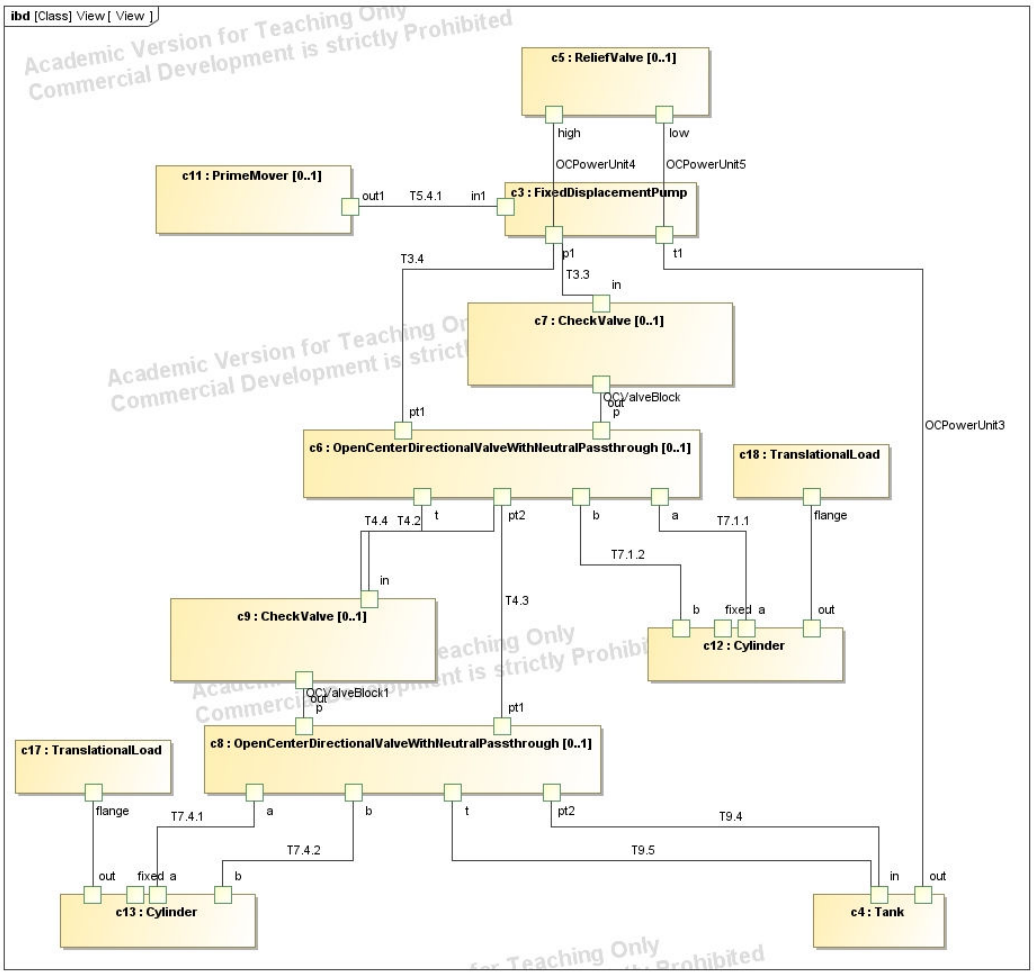

Figure 7.12: Verification example with one pump and one prime mover.

Once this example architecture was found, the next step was to add constraints to the problem and find other potential architectures. These constraints take the form:

$$\sum_{i \in D_t} (1 - d_i) + \sum_{i \in D_f} d_i \geq 1$$

where D_t is the set of indices for decision variables which are true, D_f is the set of indices for decision variables which are false, and d is an array of decision variables. Each time a new architecture is found, a new constraint is added. Then the solver is rerun

for 30 seconds. After 7 runs the two pump, two prime mover architecture is found, this architecture is shown in Figure 7.13.

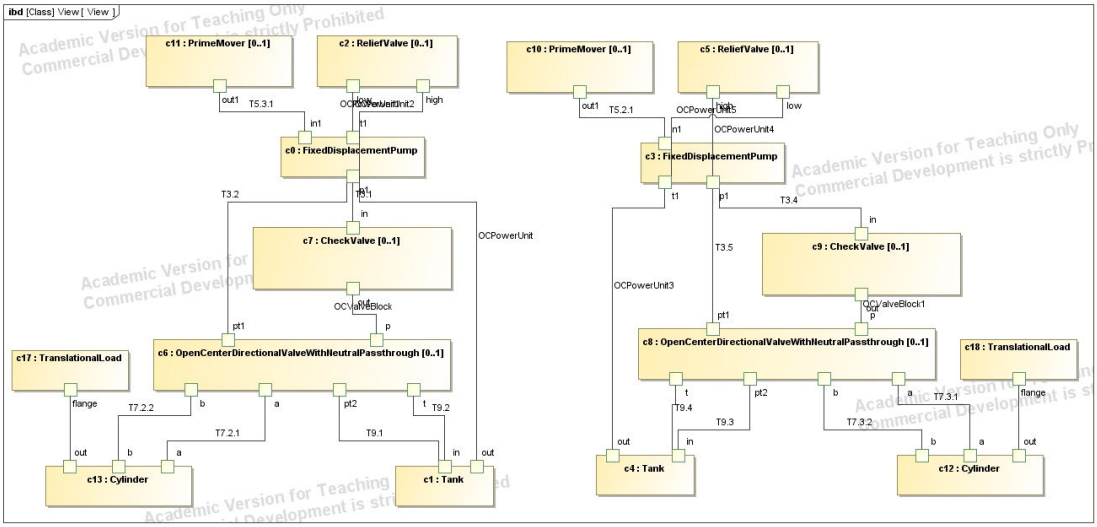

Figure 7.13: Verification example with two pumps and two prime movers.

Other configurations included 2 pumps both powered by the same prime mover, 2 prime movers powering the same pump (such a configuration would need an additional clutch between the prime movers but since here the only enforcement is Kirchhoff's laws it is not included by the solver), and both prime movers powering both pumps (this configuration would also need a clutch). The fact that the solver is able to identify these solutions implies that it is indeed searching the entire space of solutions.

To understand the effect of the approximations used during the optimization process, the optimization was rerun multiple times, each time with a different number of points as part of the interpolation of multiplications. The formulation was changed to include only one engine; this was done to simplify the determination of the relative error in the power consumption. Also, the number of component instances that were considered was decreased to improve solution times. The architecture was optimized for

fuel consumption (as grams per hour of gasoline). The expected result is that as the number of points increase, both accuracy of the interpolations and solution times will increase. The results from these optimization runs are shown in Table 7.1. The relative error refers to the relative error in the interpolation of the power provided by the engine, i.e. the relative error between the actual value of the product and the interpolated result. The power is calculated as the product of the torque and angular velocity produced by the engine. The general trend is as expected, as the number of interpolation points increases the relative error decreases will the solution time increases. When 17 points were included, the solver was not able to find the optimal before a resource interrupt after 2 hours, the best found objective value is reported. When considering the relative error in the multiplications and the change in the objective function, the error introduced by the interpolation is small. With as few as 5 interpolation points the relative error in the multiplication is less than 1%; when considering the other simplifications and assumptions made during the construction of the analysis models along with the uncertainty at preliminary design stages, the interpolation error is reasonable.

Table 7.1: Results from optimization runs where number of points in the interpallent are varied.

points	Find solution (s)	Find optimal (s)	obj value (g/hour)	# of vars	relative error
3	2.98	3.23	5071.42	1312	-0.0407
5	0.42	3.67	5200.29	1412	-0.0126
7	0.51	3.14	5151.09	1512	-0.0032
9	2.4	4.76	5125.40	1612	0.0014
11	2.78	10.68	5176.88	1712	0.0020
13	3.6	2127.18	5155.23	1812	0.00024
15	4.01	38.41	5158.85	1912	-4.4E-05
17	4.8	N/A	5347.68	2012	
19	4.99	10.48	5148.71	2112	0.00085
21	7.11	41.39	5148.41	2212	0.00038

7.5 Full Excavator Example

The full excavator example is based on the problem description in Section 7.1. The complete excavator exploration problem contains 104 decision variables, 34 for component inclusion and 70 for potential connections. This leads to 2^{104} (20,282,409,603,651,670,423,947,251,286,016 ~ $2x10^{31}$) possible combinations. Of course, this does not mean that each of these combinations represent a unique architecture, many of these combinations are symmetrically identical. In addition, not every possible combination is feasible; most of these combinations are actually infeasible, junk solutions. This is the type of problem that would be difficult to solve using black-box stochastic methods such as genetic algorithms. Considerable domain knowledge would need to be added to the mutation and cross-over operations to enable a genetic algorithm to find feasible solutions in this space.

The objective of the first optimization is to minimize cost. When minimizing the overall cost, the objective function is:

$$c_{tot} = c_{components} + c_{connections}$$

where

$$c_{components} = \sum_{components} c_{avg} \cdot d_{component}$$

$$c_{connections} = \sum_{connections} c_{avg} \cdot d_{connection}$$

The component cost is the sum of the included components and the connection cost is the sum of the included connections. Based on the structure of the architecture selection decision and previous known architectures, the expectation is for the found structure to include a single pump and single engine. The component costs can be found in Appendix A for most of the considered components, when a cost is not available the component or connection was assigned a cost of $100 dollars.

The optimization took approximately 16 hours. The best found configuration is shown in Figure 7.14. This configuration includes a single pump and engine, as expected. In addition, one interesting feature is that both boom cylinders have been connected to the same valve, as is the case with actual configurations. When looking at the sizing of the architecture, the largest pump (CPB-060) and engine (E3) in the library were selected. This is likely influenced by the amount of flow needed when all of the cylinders are moving.

In the generated solution, the engine provides power to the pump. The pump then provides flow the valves. The valves are connected in series, when a particular valve is closed the flow passes through the neutral pass through to a valve that is open (and

causing a cylinder to move). When all of the valves are open, the fluid leaving the cylinder is actually used to actuate the next cylinder. In this case, the ordering of the valves is important based on the desired priority, but this is not considered in this investigation. In addition, when the cylinder reaches the end of travel the valve should be closed to allow the flow to pass through via the neutral pass through. This does create a bit of a controllability issue, but it should be easy to design the appropriate controller based on location of the cylinder (or the human operator can simply switch the valve from on to off).

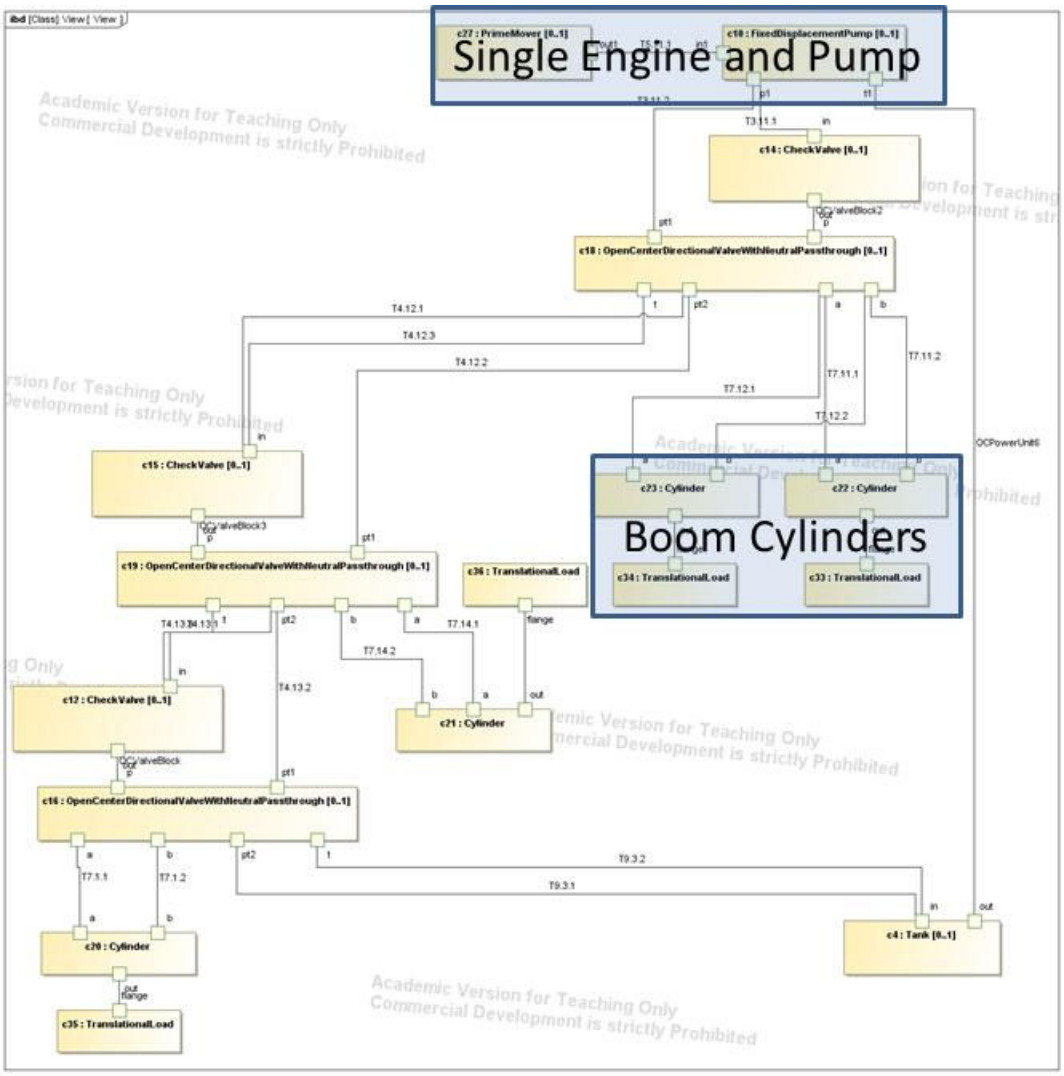

Figure 7.14: Architecture resulting from optimization of cost.

7.5.1 Optimizing for Total Cost

In the previous example, the excavator architecture was optimized for component cost. The next optimization performed was to minimize the life-time cost, which includes both the component cost and fuel costs. This objective function can be states as follows:

$$c_{total} = c_{tot} + fuel_{total} \cdot \$4 \cdot 10{,}000\ hours$$

where c_{tot} and $fuel_{total}$ are the quantities from the previous section. The units on the fuel consumption rate are chosen so that the fuel total is in gallons per hour. The brake specific fuel consumption refers to the weight of the fuel consumed, so the specific gravity of gasoline (719.7 kg/m^3) is used to perform the conversion The final total is then converted from m^3 into gallons (264.17 gallons/m^3). The fuel cost is estimated to be $4 dollars a gallon based on current prices. Also, the average lifetime of a machine is approximately 10,000 hours.

The optimization ran for approximately 36 hours at which it was ended due to a resource interrupt. This interrupt was caused by the solver hitting the maximum number of iterations. The found solution is illustrated in Figure 7.15, with a single prime mover powering multiple pumps. Again, the boom cylinders have been grouped together. The boom cylinders are powered by one fixed-displacement pump and then the arm and bucket are powered by another fixed-displacement pump. Since this is the result after a resource interrupt, the solver only asserts that this is the best solution found during the search. In previous optimizations, minimizing the fuel consumption would push the solution toward one with multiple pumps while minimizing cost would push the solution toward one with a single pump. It he solver has selected a compromised where two pumps instead of a possible four are included.

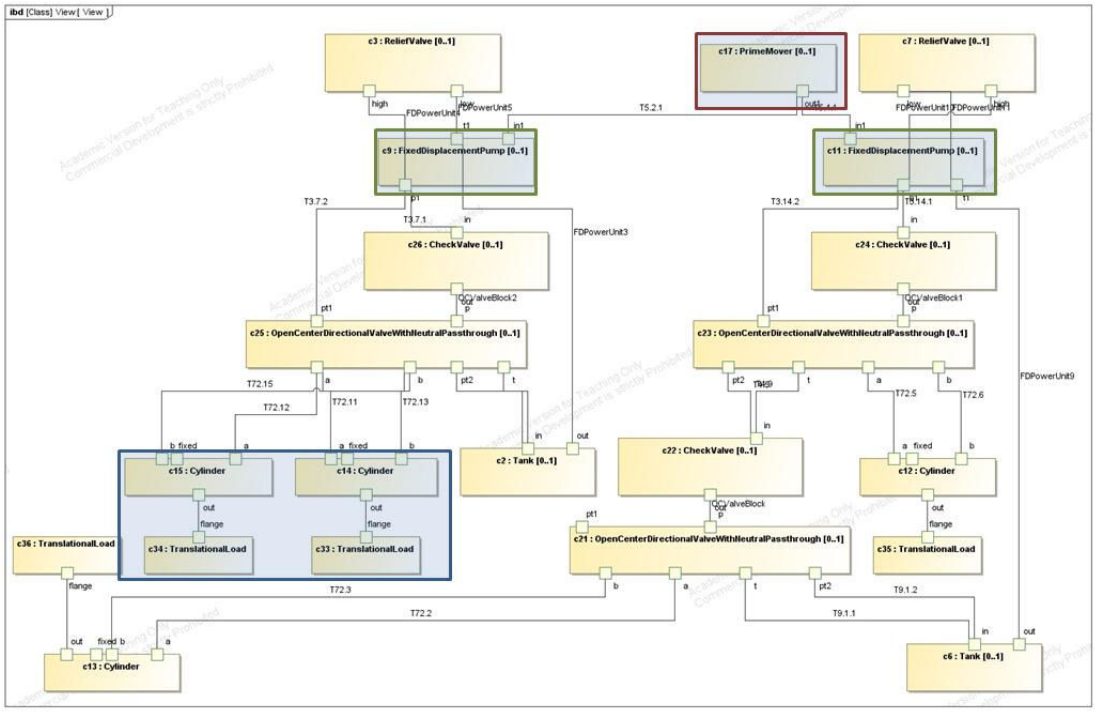

Figure 7.15: Architecture found when minimizing life-time cost.

An optimization that takes 36 hours is a significant computational investment, even if computational resources are cheap. In the next sections, how to manage this overall computational cost is considered along with how this compares to other methods.

7.6 Constrained Example

To demonstrate one potential avenue for managing scalability, a constrained version of the hydraulic subsystem selection decision is considered in this section. Instead of allowing up to 4 different pumps and 4 different prime movers, instead only 2 pumps and 2 prime movers are considered. In addition, the connections between the valves and cylinders are fixed.

Although significantly more constrained than the previous version, this example is more consistent with the types of explorations that would be performed in practice. In current design practice, it is unlikely for engineers to completely redesign an entire

architecture from scratch without reusing any previous structure. Instead, an alternative approach is to fix most of the architecture and consider varying only the most important aspects.

This greatly reduces the number of solutions that need to be considered, but even though the exploration process is simplified it is still a complex and interesting example. Even with the constrained design space, there are still 2^{50}(1,125,899,906,842,624 ~ 1 x 10^{15}) combinations.

The structure of the constrained example is shown in Figure 7.16 and Figure 7.17.

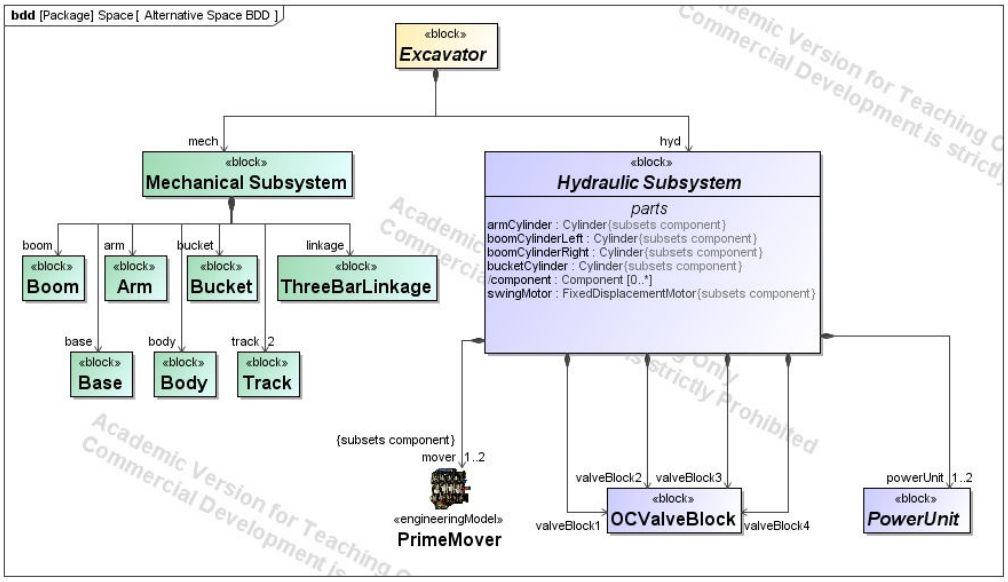

Figure 7.16: Structure of the constrained example.

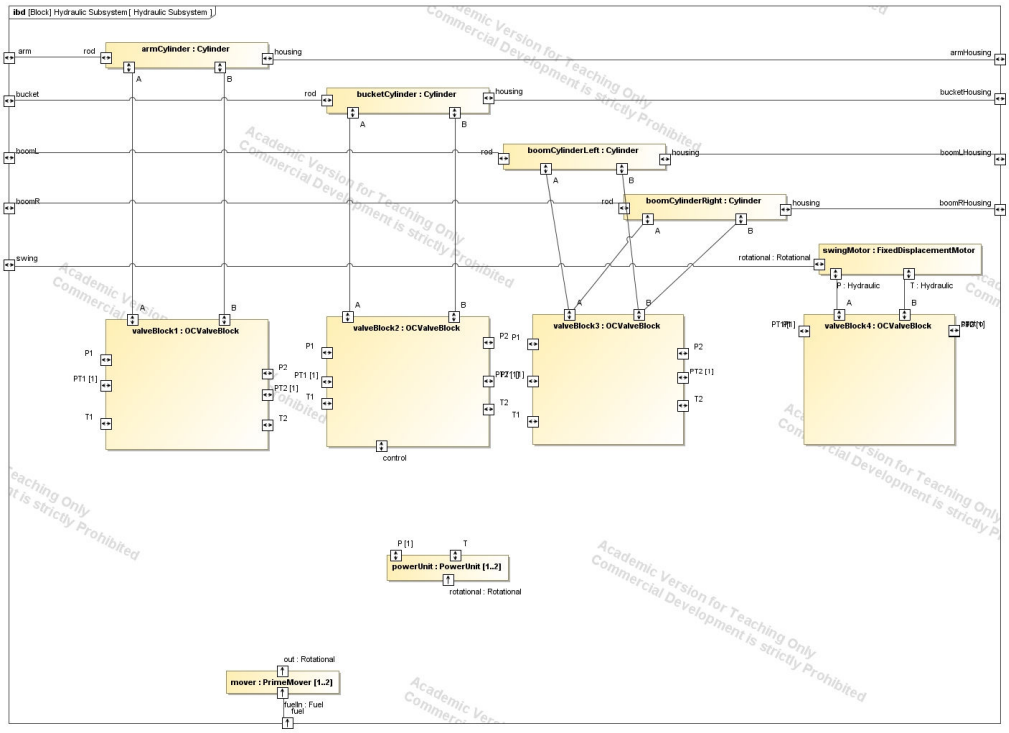

Figure 7.17: Internal structure of the constrained example.

Two different optimizations are performed with the constrained example, one where the overall cost of the system is minimized and one where the fuel consumption of the system is minimized. The objective function for cost was discussed in Section 7.5.

The best architecture found after ~2 hours of optimization time is shown in Figure 7.18. The optimization times are provided as approximate estimates because CPLEX is run in opportunistic mode where randomness is included in the optimization. As expected, there is a single engine powering a single fixed displacement pump.

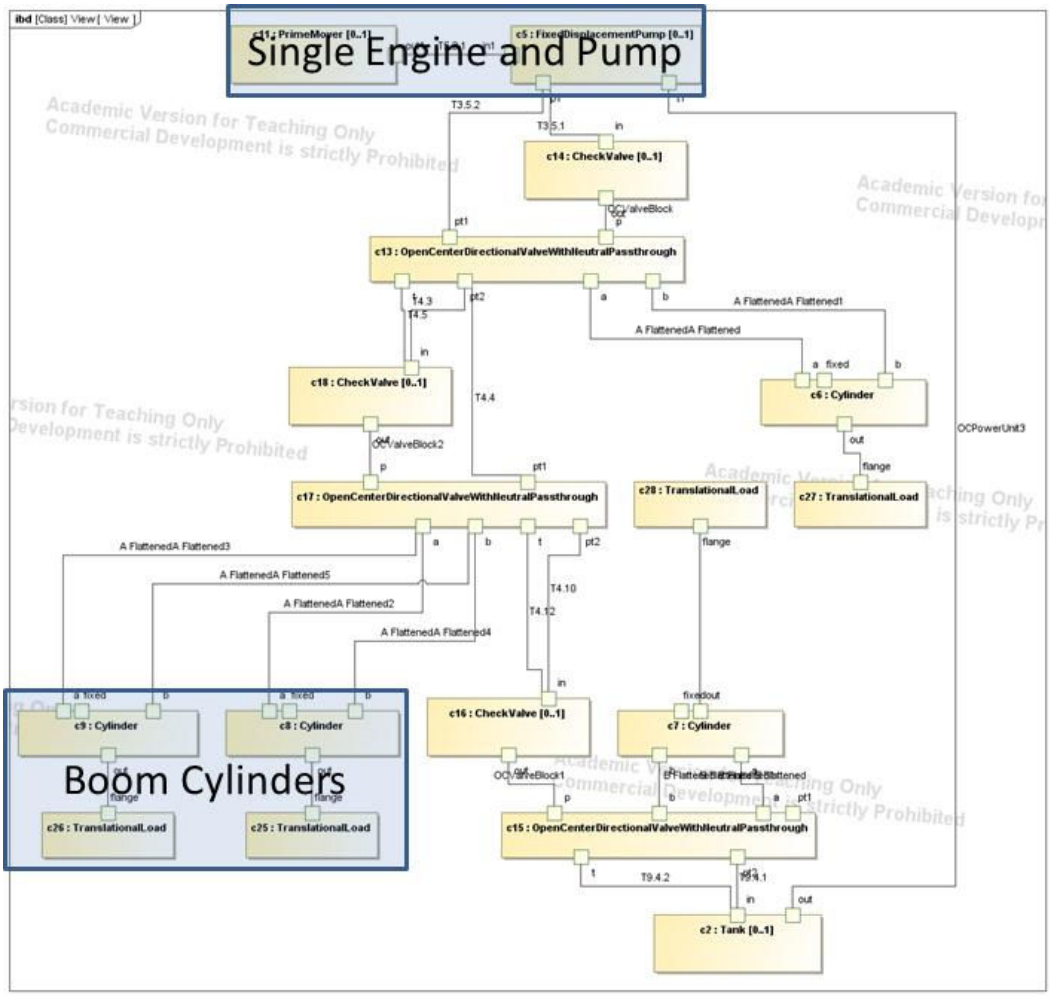

Figure 7.18: Architecture found when minimizing cost.

The second optimization is in minimizing fuel consumption. The objective function is:

$$fuel_{total} = \sum_{s} \sum_{engines} fuel \cdot t$$

where s is the number of scenarios, $engines$ is the set of engines, t is the time of each scenario (the assumption is each scenario takes an equal amount of time), and $fuel$ is the amount of fuel used by the particular engine during that particular scenario. Unlike the minimize cost example, the best architecture is not as apparent. Previous work in hydraulic architectures suggests that the best configuration will use multiple pumps; also the structure of the pump equations suggests that minimizing the pressure inside each pump reduces the losses for that particular pump. This also suggests that multiple pumps are more fuel efficient. For the use scenarios, the structure of the equations would also suggest that multiple smaller engines would be more fuel efficient than a single larger engine.

The found architecture after approximately 5 hours of optimization is illustrated in Figure 7.19; as expected this architecture includes both multiple pumps and engines. The pumps are highlighted in green and the engines in red.

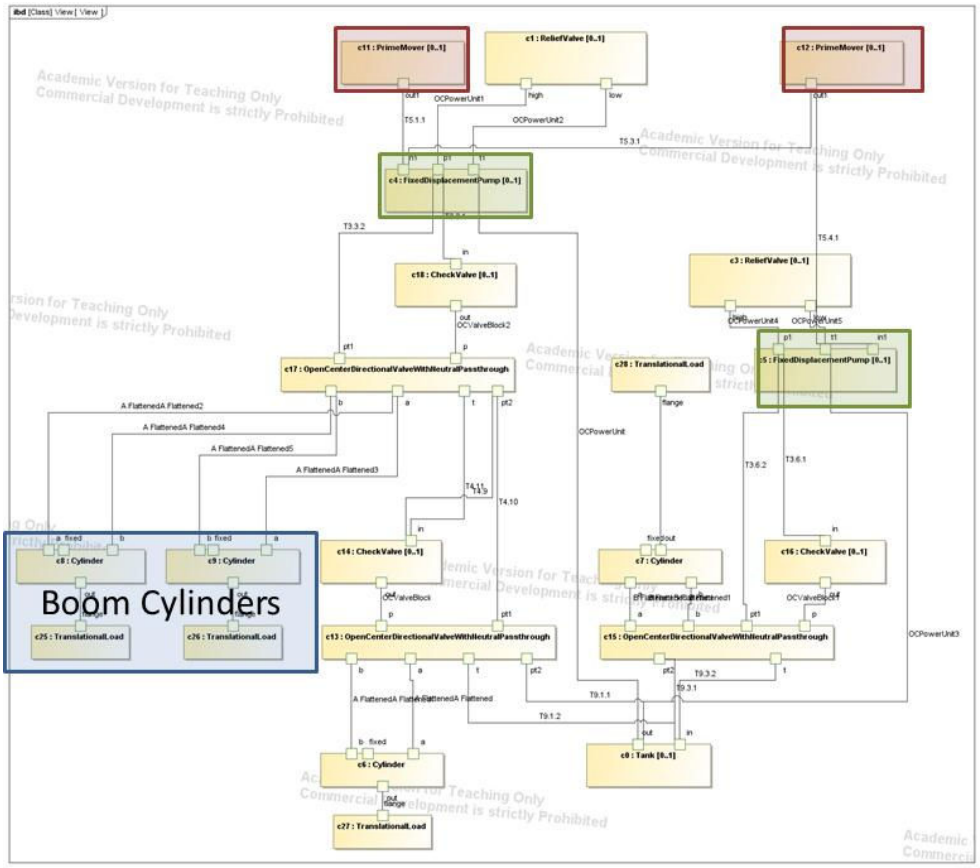

Figure 7.19: Architecture found when minimizing fuel consumption.

These solution times are significantly smaller than in the previous excavator example.

7.7 Unsized Solutions

Another potential avenue to decrease the computational cost is to generate unsized solutions and then size them in a subsequent step. This avenue is not extensively explored in this investigation, but the capability is demonstrated with further investigation left as a possible extension. With unsized solutions, the analysis of the behavior can be significantly simplified, which also simplifies the constraints used in the formulation. Instead of needing to accurately choose variables related to pressures, flows,

losses, and so forth, these variables can generically capture whether flow is or is not present or high pressure is available. One feature of the presented framework is that it can also be used for generating unsized solutions. New algebraic models can be created by using the existing models and then simplifying or removing appropriate algebraic constraints. There is the potential for this process to function automatically, but in this case it was done manually. Any sizing variables were removed from constraints, losses and constraints related to losses were removed, and the torque interpolations in the engine were also removed to capture that the engine could produce torque.

These new models were used to optimize the two cylinder verification example for cost. The architecture in Figure 7.14 was found as the optimal solution in 5 seconds. In addition, CPLEX supports the generation of a solution pool which can contain multiple solutions. This optimization was enabled for the example and an additional 500 solutions were generated in less than 30 seconds. Unlike the previous example of generating multiple architectures in Section 7.4 where constraints are used to prevent the solver from selecting the same architecture, with the solver's current capabilities it is not possible to specify that each of these solutions should have a different architecture. Among the first 50 solutions, 10 different architectures were present (although not all of these are unique). Further investigation is needed into this area to efficiently generate multiple architectures, but it holds significant promise.

7.8 Comparison with Similar Approaches

This section provides a decision on the benefits of the presented method over current state-of-the-art computational design synthesis approaches and also the limitations in regards to those approaches. In Chapter 2 and Chapter 5, a broad listing of a

number of different solution approaches is provided. A detailed comparison to every computation design synthesis or architecture exploration approach is impossible because both are active and ever expanding fields, each with a huge number of potential approaches. Instead, this comparison will highlight two approaches: a constraint-based approach by Wynn et al (Wyatt, 2012) with a flexible schema for defining the constraints for a particular problem, and a multi-stage genetic algorithm based approach by Pederson (Pedersen, 2007) where more domain-specific knowledge is captured (this approach is more directly focused on optimizing hydraulic systems).

This comparison is informed by the examples presented in this chapter. Although the examples from this chapter are different than those presented in the other two approaches, the commonality in terms of goals, the structuring of system architectures, and system size allows for comparison.

The first considered approach will be labeled the Wynn approach. In this approach, a designer is free to specify the space of architecture by specifying the potential components and relationships between these components. This is very similar to the approach presented in Chapter 3, except in the Wynn approach the relationships are defined between components whereas in this approach connections are defined between interfaces. In some ways this is a minor semantic difference, but in this approach specifying the connections between interfaces allows the inclusion of analysis knowledge.

To further define the search space in the Wynn approach, network structures constraints are used to define what relationships and components exist in feasible

configurations. These network structure constraints apply directly to the structure of a particular architecture. The constraints included are as follows:

- Component number constraint: limits the number of each particular component that can be included in the architecture.
- Direct connection constraint: requires a direct connection between two particular component types.
- Fan out constraints: requires that a particular component has a certain number of incoming or outgoing constraints of a certain type.
- Indirect connection constraint: requires a path between two particular component types, but multiple components and relationships can be present for this path. The exact number of intermediate elements can also be constrained.

The first three constraint types are easily defined in MIP or in a SAT based approach, but the fourth type of constraint is more difficult to define. As a result, in the Wynn approach, the search process starts at a known solution (or will attempt to generate an initial guess by choosing the minimum number of each type of component and relationships), attempts to mutate that solution by adding components or connections, and then checks if the mutated solution is still feasible. In the prior work, the approach is shown to be capable of generating feasible architectures for several mechanical systems. These feasible architectures then serve as the input to the next stage of the design process where designers could manually prune the results and begin the sizing process. Since only an architecture's structure is considered, complexity metrics are used to sort potential architectures, for instance the number of components and connections.

This approach has two interesting characteristics. First, the modeling language for representing the design space is tailored specifically to representing only components, relationships, and the types of constraints described in a simple visual form. This makes the approach easy for designers to use because they only need to learn a few constructs and the complete definition of a design space is relatively small. Second, since the solver is specifically designed for architecture exploration it is easy to identify unique architectures and store these feasible architectures during the search process.

The tailored modeling language makes it significantly easier for designers to capture their design space in the Wynn approach than with the language described in this investigation. This can be attributed to two important factors: in the Wynn approach the authoring tool is specifically designed for the language, and in the Wynn approach less knowledge is encoded as part of the design space definition. On the other hand, the modeling language in Chapter 3 has the advantage of being based on a standardized language, SysML, which allows this approach in this investigation to be more easily integrated with other MBSE efforts. Also, an authoring tool could be constructed specifically for the modeling language in Chapter 3 instead of SysML in general. This tool could force the user into a workflow consistent with the architecture selection decision definition presented in Section 3.4 and could reduce the number of constructs the user would need to understand.

The other consideration is could the modeling language in Chapter 3 be simplified to include less designer knowledge and still be adequate. This consideration is more difficult to address, because identifying whether the constructs selected indeed make up the simplest version of the language is not considered in this investigation. However, the

excavator example does illustrate a significant shortcoming in the Wynn approach and suggests that more designer knowledge is needed as part of the formulation.

One important characteristic of the excavator architectures identified by the solver is that the two boom cylinders are connected to the same valve while the other cylinders are connected to their own valve. Capturing this distinction using the Wynn approach would be difficult (although this structure could be directly encoded and enforced, which would work for this example because the structure is well known, but in general would over-constrain the design space). Specifying that any valve could connect to up to two cylinders would require designers to prune a huge number of (actually) infeasible architectures after the solution process finishes. The reason that the approach in this investigation can identify that the boom cylinders should connect to the same valve is that multiple use scenarios can be included as part of the definition of the architecture exploration problem and how the components operate is also captured and taken into account. This suggests that although the Wynn representation is significantly simpler, it does not include all of the knowledge needed for effectively modeling architecture exploration problems.

In addition, the Wynn approach lacks a method for handling architecture sizing. For the resulting feasible architectures that are identified by the search process, analysis models would be needed to analyze and sizing these architectures. Either these would need to be defined manually, or a composition approach similar to that in Chapter 4 would be needed. If an automated approach is used, designers would need to encode additional analysis knowledge in a way that is very similar to the approach in this investigation.

The previous factors suggest that although the Wynn approach is simpler, this approach is more comprehensive. That being said, the choice of best approach is dependent on the architecture exploration problem being considered. For simpler problems, the lower modeling cost present in Wynn's approach could make it more desirable. On the other hand, for problems where the modeling of use scenarios is necessary (such as the excavator example), or system sizing is important, the approach in this investigation would be more desirable.

The second approach considered is labeled the Pederson approach. In this approach, the designer can represent search spaces which are composed entirely of hydraulic components. The representation schema is geared specifically to hydraulics, although interfaces are only implicitly represented. Different use scenarios can be also be included. Designers cannot encode their analysis knowledge in the framework; instead the analysis knowledge is hardcoded a priori. In some ways, this is a facsimile of the model libraries considered in this investigation. That being said, there are multiple types of analyses included, for instance cost, noise, and so forth. An objective function can be defined that combines the results of these analyses in a multi-attribute objective function where the attributes are weighted and then summed together. (As an aside, the goal of this investigation is to support rational decision making. The inclusion of such a multi-attribute objective function can lead to inconsistent results if preferential independence is not established, but this is a discussion best left alone in this investigation).

In order to find optimal solutions, Pederson uses a multi-layered genetic algorithm based-approach where architecture selection parameters are modified by a genetic algorithm in the first layer, and sizing is done by tailored optimization approaches in the

subsequent layers. Since only hydraulic architectures are considered, the subsequent optimizers can be (and are) specifically tailored to accurately and efficiently size a particular architecture.

When comparing to Pederson's work, the overall approach is significantly different. Pederson's approach is specifically tailored to hydraulic systems, whereas this approach is more general although applied exclusively to hydraulic examples.

The large example provided in Pederson's work is the optimization of the hydraulic architecture of a forklift. This example is very similar to the excavator example provided here, it has four actuators that need to be supplied with hydraulic fluid. Unlike the excavator, only one motor is available to power the system.

Tailoring the schema for hydraulic systems does have the advantage of making the schema easier to use and unlike the Wynn approach the same types of designer knowledge are captured in the Pederson approach and the approach presented in this investigation. In the Pederson approach, significant effort was invested in encoding and verifying the analysis knowledge included. This is especially necessary when it is difficult for users to change their analysis knowledge. It also makes it difficult to include new technology in the problem formulation.

In this investigation, all of the designer knowledge included in the architecture selection decision is represented using the same modeling language. Designers are free to adjust any aspect of this definition, including the analysis knowledge or types of components. When new technology emerges, it can be added to the definition of an architecture selection decision as long as its behavior can be modeled in a representation

that is consistent with the method described in Chapter 5, i.e. using mixed-integer linear algebraic constraints.

The other major difference between this approach and the Pederson approach is in the search process. Pederson uses well known optimization algorithms which are then hard-coded as part of his framework, in general these are fairly simple algorithms that are tuned for sizing hydraulic systems. In this approach the goal is to transform the definition of the architecture exploration problem into a form which can be understood by more sophisticated solvers.

It is difficult to truly compare the two methods because the Pederson approach relies on black-box simulation models to describe the dynamics of the system. Unlike this approach, these models are both static and pseudo-dynamic (they statically approximate dynamic behavior using finite differences), whereas in this approach only static behavior is considered. Also, in Pederson's approach a greater number of hydraulic components were considered, including closed center architectures that include both fixed displacement and variable displacement pumps.

The more accurate analyses that Pederson performs may be useful for inclusion in this framework after the initial architecture exploration phase. Since Pederson does not report solution times, it is difficult to compare the two solution approaches. Since the solvers used in this approach can explicitly account for the structure of the problem, it is likely that they are more efficient. There is anecdotal evidence for this statement, in Pederson's verification examples, the genetic algorithm only runs for 10 generations and convergence is not demonstrated. This suggests that the solution process is very computationally expensive, and that the solutions found by the genetic algorithm are

suboptimal. In addition, since the approach relies on the algorithm to mutate existing feasible initial guesses, it is likely that the solutions are found near previous solutions and a true exploration of the space is not being performed. Pederson does not present any evidence that the solution approach is actually able to move around the space, and the constrained nature of hydraulic systems raises concerns that the genetic algorithm is not able to move from one feasible solution to another.

7.9 Summary

In this chapter, the architecture selection of the hydraulic subsystem for an excavator is presented. Before the full selection is presented, several smaller verification examples were presented in Section 7.4. The goal of these verification examples was to establish the capability of the mathematical programming solver to find solutions for the formulation presented in 7.3 and also to demonstrate the solver could indeed identify the interesting solutions in the space. This is done by running the optimization multiple times, each time adding constraints to eliminate previously discovered architectures. Once this was established, the full excavator example was presented along with a constrained version. The constrained version was presented to address one potential method for managing problem scale, namely restricting the design space but restricting it in a meaningful way so that the results were still interesting to the designer. Using the synthesis of unsized architectures was also considered as a ways to mitigate scalability concerns. Also, in Section 7.8, the approach in this investigation was compared to other similar approaches.

CHAPTER 8:

CONTRIBUTIONS, LIMITATIONS AND OPEN QUESTIONS

This chapter is a review of the material from the previous chapters. The main objective is to reexamine the research questions and hypotheses and identify the insights gained from the results presented in the previous chapters.

8.1 Review of the Research

The broad motivation for this research was expressed in Chapter 1 in the following research question:

How should designers best represent, manage, and apply knowledge for efficient exploration of system architectures?

The research objective was to study a particular approach to representing and performing architecture explorations which was outlined in Section 1.6.

In this approach an architecture exploration problem is represented as an architecture selection decision using an information modeling language. This representation is then transformed to generate solution specific formulations that can be interpreted by various solvers. The focus is specifically on generating corresponding mathematical programming optimization problems which can be solved by mathematical programming solvers.

The overall research question was broken down into four more manageable questions that need to be solved before the overall question can be addressed. These questions are as follows:

RQ1. How should the designer define an architecture exploration problem?

This question was address by hypothesis 1:

H1: Designers can represent their architecture exploration problem in information models as an architecture selection decision consistent with decision theory using a domain-specific language.

The main evidence in support for this hypothesis is as follows:

- The literature in decision-based design establishes that the design process can be modeled as a set of decisions and decision theory can serve as a prescriptive framework for designers. As discussed in Chapter 2, the current selection methods used for architecture selection decisions are very ad hoc and can lead to inconsistent and self-contradictory decision making. On the other hand, decision theory provides a prescriptive approach which if followed is guaranteed to result in decisions which are consistent with a decision maker's beliefs and preferences.
- The literature also establishes Model-Based Systems Engineering as an emerging trend in the systems engineering community. In MBSE, information models form the basis for documenting the artifacts produced during systems engineering processes. This suggests that information models are a good starting point for representing the architecture exploration problem.
- Based on this prior work, the justification for modeling an architecture exploration problem as an architecture selection decision is presented in Section 3.2. This justification is based on previous work in decision-based design where engineering design is represented as a set of sequential decisions.

- An information modeling language for representing architecture selection decisions is presented in Section 3.4. This modeling language is a domain-specific language that extends the established and standardized SysML language, demonstrating the concepts necessary for
- The modeling language is used to define the architecture exploration problem for the hydraulic excavator in Section 7.1. This demonstrates that the modeling language can be used to represent architecture exploration problems of some scale.

RQ2. How can domain-specific synthesis and analysis knowledge be captured and organized effectively to allow for composition and reuse?

This question was addressed by hypothesis 2:

H2: Designers could use modularity and composition along with model transformations to reuse knowledge encoded in models.

The main evidence in support of this hypothesis is as follows:

- The literature establishes that analysis models can be composed into more complex analysis models via well-defined interfaces as long as they are imperative analysis models. A review of the related literature is provided in Section 4.1.
- In Section 4.2, an approach for capturing reusable fragments in model libraries is presented. This approach is based on storing composable model fragments in model libraries and tagging these fragments with meta-data in the form of aspects.

- One enabling characteristic of the composition process is the use of explicit relationships to capture the correspondence between component-level structural models and component-level analysis models. This process is described in Section 4.2.4.
- In Section 4.3, how the model libraries can be represented in SysML is described. Without the ability to represent both the architecture selection decision and the relevant model libraries in the same language, it would be more difficult to represent the connections described in Section 4.2.4.
- Transformation approaches are demonstrated for generating two different types of analyses, a dynamic analysis in Chapter 4 and the mathematical programming optimization formulation in Chapter 6 and 7. These transformation approaches operate on SysML models which conform to the language definition in Chapter 3 and 4.
- The transformation approach for mathematical programming is verified in Section 7.4 by using it to generate AIMMS code based on the presented examples. This code is reviewed to insure that it matches the expected result and is also used to identify the architectures found in this section.
- The transformation approach is used to generate the AIMMS code used for the Excavator example in Section 7.5, this demonstrates the scalability of the approach. Also reusing the same models between different examples illustrates the potential for reusability.

RQ3. What optimization framework is best suited for identifying promising architectures?

This question was addressed by hypothesis 3:

H3: Designers could use mathematical programming techniques to identify promising solutions early in the exploration. Mixed-Integer Linear Programming should be used for architecture selection.

The main evidence in support of this hypothesis is as follows:

- The literature establishes mathematical programming as potentially applicable to architecture exploration with one important consideration being the availability of high-quality commercial solvers.
- A modular approach for representing an architecture selection decision within a mathematical programming language as a mixed-integer linear programming problem is presented in Chapter 5. This description is based on the structure of an architecture selection decision presented in Section 3.3. This approach provides a framework for representing architecture selection decisions in mathematical programming terms.
- In addition, some discussion on how to represent nonlinear behavior as linear constraints is described in Section 5.4.3. Without being able to represent nonlinear behavior it would significantly restrict the analysis knowledge that could be encoded in the framework and therefore would significantly reduce the accuracy of the initial exploration step. Also, this would likely make the framework impractical for architecture exploration problems were nonlinear behavior is important to the selection of a candidate architecture.

- Example problems are provided in Chapter 7 where CPLEX, a mixed-integer linear programming solver, is used to identify promising solutions based on a particular objective. The example problems demonstrate several capabilities of CPLEX, including the ability to identify feasible solutions, identify promising solutions, and create solution pools.
- The mixed-integer linear programming results are compared with other similar approaches in Section 7.8. Although the results presented in this investigation do not conclusively demonstrate that mixed-integer linear programming and mixed-integer linear programming solvers are always the best approach, the results do show that it is applicable to sizeable problems and compares favorably with other methods.

RQ4. How should problem scale be managed?

Unlike the previous research questions, this research question is not directly answered with a hypothesis. Throughout this study, how problem scale should be managed was central to the choice of relevant technologies or the formulation of a particular approach. The practices identified in this work can be summarized as follows:

- The object-oriented nature of the modeling language for architecture selection decisions simplifies the representation of the decision because inheritance, redefinition, and usage concepts can be used.
- Model libraries are used to capture reusable model fragments which can be used when specifying a particular architecture selection decision. This simplifies the definition of subsequent problems once the model libraries have been created.

- Connection templates allow designers to group together common connections types, instead of assigning a decision variable to each connection this allows a single decision variable to be assigned to the entire connection template. This reduces the number of variables in the resulting mathematical programming formulation, which reduces the number of combinations the solver must investigate.
- The same is true for the inclusion of functional units/subsystems which combine together components and their connections into well-established groupings. This has two effects, instead of requiring a designer to include all of the components and connections in the description of the architecture selection decision, only the functional unit needs to be included. Also, the number of decision variables related to the functional unit greatly decreases because of instead of requiring a decision variable for each component and potential connection, the entire grouping can be related to one (or a small number if there are optional components) decision variable.
- Another major simplification made is to use only linear equations in the mathematical programming formulation. Linear solvers are usually able to handle much larger mathematical programming problems.
- In Chapter 7, it was demonstrated how a potentially unwieldy problem can be scoped by reducing the search space and constraining parts of the architecture.
- In Chapter 7, it is also demonstrated how the same framework can be used to quickly generate a space of unsized solutions which could be sized using more conventional techniques.

- The scaling approach for interpolation outline in Section 5.4.3 reduces the number of data points (and therefore the number of variables) required for each interpellant reducing the size of the problem.

8.2 Summary of Contributions

8.2.1 Modeling Architecture Exploration Problems

The use of information models in systems engineering is gaining popularity, especially with the continued adoption of Model-Based Systems Engineering. The goal of this research is to extend the basic scope of MBSE to also include supporting decision making during these processes. The current state of the art is focused on the documentation of systems engineering problems and processes in information models.

As discussed in Chapter 2, current systems engineering processes are very ad hoc; they rely largely on previous experience and qualitative metrics to steer the design of the system at early stages. Facilitating quantitative evaluation at these early stages provides a significant tool not currently available to designers. Also, the explicit representation of the architecture selection problem within information models is a first step toward more rational design processes at early stages of system design. As discussed in Chapter 3, decision theory is only applicable when there is a single decision maker, and when multiple decision makers are present there is no rational approach for aggregating their ordering of potential alternatives without the presence of a dictator (i.e., a single decision maker situation masquerading as a group decision). By providing a team of designers an explicit information model where beliefs can be recorded and consensus reached, there is the opportunity for more rational decisions than previously possible by approximating a single decision maker situation. For instance, this could allow the aggregation of

multiple designers' beliefs about outcomes, which could then be rank ordered by a single decision maker which would be consistent with decision theory.

In addition, this research has demonstrated in a new potential application of SysML. Although SysML has been previously used in an extensive array of applications, most of these have been driven by documenting existing artifacts of the design process. The SysML representation of an architecture selection decision is a significant departure from these earlier goals, but does fit in with work by the INCOSE MBSE Model Management Working Group to include the definition of variants as part of the SysML language. The results of this investigation should inform that effort to standardize the definition of spaces of potential solutions, specifically in terms of the need to represent potential connections.

8.2.2 Architecture Exploration and Computational Design Synthesis

The representation of the space of potential architectures has important implications within the domain of computational design synthesis. The concept of representing the design space in a modular fashion where composition can be used to generate analysis models and simulations can be used to support and improve most architecture exploration or computational design synthesis approaches.

When looking at past approaches, one limitation is the failure to include of both the synthesis of potential alternatives and the analysis of these alternatives in a single framework. As discussed in Chapter 1 and Chapter 2, many previous methods rely heavily on encoding knowledge only about allowable system structure and use only this knowledge to synthesize alternatives. Although it is important for designers to be able to encode this knowledge about the structure of a potential system, without an approach to

analyze potential systems it is difficult to distinguish between promising, feasible, and infeasible solutions by only considering the structure. Designers are simply implicitly encoding their previous experience and beliefs about how the system operates instead of explicitly encoding it as analysis knowledge. On the other hand, inclusion of analysis knowledge also is based on previous experience and beliefs, but the argument here is that knowledge how to predict system behavior is less susceptible to bias.

In addition, the framework presented can both identify feasible (or promising) architectures (what are the components, how are they connected together?) and also initial sizings for each of the components in the architecture. The fact that both architecture selection and component sizing is handled in the same framework (using the same analysis knowledge) is a departure from previous frameworks where the identification of potential architectures is separate from sizing those architectures. The advantage of this approach is that the more analysis knowledge included in the representation, both the selection of potential architectures and the selection of component sizings become more accurate.

8.2.3 Mathematical Programming

This work has also further established the relevance of mathematical programming to the domain of mechanical design and systems engineering. One of the major hurtles to the wide-spread adoption of mathematical programming techniques (and the use of the existing, high-quality commercial solvers that are available) is the difficulty for engineers and designers to represent their knowledge and problem descriptions in mathematical programming. From the examples in Chapter 7, the mathematical programming problems often contain thousands of constraints and variables. Even simply

expressing these in a visual or object-oriented fashion is insufficient. In this research, many of these constraints or variables are implicitly defined in the information models because they are at a higher level of abstraction; this is a conscious decision to abstract away these constructs for the convenience and ease of use of a designer. This demonstrates the benefit of creating high-level more abstract constructs to capture this knowledge, and then transforming these more abstract representations into low-level mathematical programming languages.

8.3 Limitations

There are several limitations or caveats associated with the presented approach. The following is a summary of the most notable.

8.3.1 Cost of Modeling

As discussed throughout this thesis, one of the major cost drives is that the explicit modeling of the architecture selection decision comes at a higher initial time investment than previous methods. One of the major advantages of previous document approaches is the accessibility of design documents, only minimal training is needed to understand and create these documents. With information models, designers and engineers need additional training in authoring tools, such as MagicDraw (No Magic Inc., 2012), and in the SysML language.

A company adopting MBSE principles will have a significant initial investment in workforce training (and to a lesser extent the appropriate tools). In order for MBSE to be a value-added endeavor, the cost of this initial investment must be offset by the gain in future productivity or in the overall efficiency or effectiveness of the design process. This can take many forms, for instance although there is significant initial investment, model

transformations could be used to automatically generate analyses later in the design processes which otherwise might need to be created by hand. In this investigation, how to reduce the modeling cost between projects is demonstrated through reuse. Also, there is a significant reduction in design effort by using composition to generate various analyses needed during the design process.

8.3.2 Creating Model Libraries

While employing model libraries and reusing fragments from these model libraries can significant reduce the modeling cost for a particular problem or project, creating these libraries does require significant upfront investment. In order for the potential to reuse these libraries to exist in a practical context, the knowledge included in the libraries must be in a form where it is applicable to a wide variety of potential situations but also extensively verified. One of the drawbacks of using the constraint-based approach as described is that an incorrectly formulated library model can cause problems with the entire search process.

8.3.3 Uncertainty

Design decisions are not made with perfect knowledge; there is a significant amount of inherent uncertainty throughout the design process. In the presented framework, the discussion of uncertainty is largely neglected because the use of deterministic instead of predictive analysis models greatly simplifies the definition of the analyses and objective function.

This leads to a significant fundamental issue: is it appropriate to only consider uncertainty implicitly when the magnitude and effect of uncertainty likely has the most effect of any point in the design process? The other consideration is how this work could

be extended to include uncertainty, and whether this inclusion will significantly change the structure of the framework.

8.3.4 Scalability

As with any architecture exploration approach, a primary concern is scalability. The problem with discussing "scalability" is that the scale of a problem has many dimensions. In this work, the scale of a particular architecture selection decision has been described by the number of constraints and variables that are present in the mathematical programming formulation.

The larger the mathematical programming formulation becomes in these terms, the more difficult it is for the solver to find feasible or optimal solutions. This difficulty usually translates into longer solve times; if these solution times become unmanageable then the value added is significantly decreased.

The other issue is the limitation on CPU power and memory availability. The CPLEX solver stores the search tree in memory, as this tree grows in size so does the memory footprint. For some of the experiments run in Chapter 7, the tree's footprint could grow to be as large as 8 gigabytes. This is partially offset because the cost of computational resources continues to decrease. Additional investigation into how the problem can be rationally decomposed may be a better way to address this issue.

8.3.5 Accuracy of Analyses – Using Only Linear Constraints

Another issue that is strongly tied with scalability is the accuracy of the analyses used during the exploration process. The analyses used in this investigation are based on linear algebraic constraints that approximate steady-state behavior; while this was adequate for the examples provided in Chapter 7, there are other domains or other types

of exploration problems were more accurate analyses or more comprehensive analyses are desirable.

In the examples in Chapter 7, the analyses focused on approximating fuel consumption and component costs. Fuel consumption is an important factor in the current consumer climate because of increasing fuel prices and also more stringent environmental regulation. That being said, there are other contexts where the important technical characteristics of a design may have more to do with reliability, some other measure of efficiency, or overall functionality/performance. In this context, the functionality of the system being designed is well-known and if a potential architecture is unable to accomplish the basic functionality it is considered to be a poor solution which is discarded. The addition of other analyses to the mathematical programming formulation means the inclusion of additional constraints and variables, which makes the problem more difficult to solve.

The inclusion of more accurate analyses has the same effect. One of the major assumptions is that the behavior being modeled can be approximated using linear constraints. In Chapter 5 an interpolation approach for approximating nonlinear behavior was presented, but the accuracy of this approximation is a function of the number of data points included in the interpellant. With each additional data point comes additional variables that the solver must consider. Also, with the inclusion of more complex nonlinear behavior, the number of interpolations will rise. Overall, this will increase the size of the mathematical programming problem making it more difficult or potentially impossible to solve.

8.3.6 Non-Unique Architectures

In this investigation, the representation for a particular architecture Is not unique. In both the SysML formulation and the mathematical programming formulation, an architecture is described by its set of components and connections. Each component is assigned a particular binary variable and each connection is assigned a particular binary variable, if these binary variables are true the connection or component is included in the architecture. The union of all the binary variable values describes a particular architecture.

The issue is that the same architecture can be described by different unions of binary variables. For example, if the design space restricts architectures to a maximum of 4 pumps, then a different binary variable will be created for each pump:(x_1, x_2, x_3, x_4). An architecture with 2 pumps could be represented as $(0,0,1,1)$ or $(1,1,0,0)$ and so forth.

This becomes a significant issue during the solution process; the solver must either search through a large number of identical architectures or intelligently identify the symmetry in the structure of the constraints and eliminate redundant nodes from the search tree.

Another way to address this issue is to constrain the formulation so that these identical architectures are not included or can be quickly eliminated by the solver.

8.3.7 Debugging the Formulation

As is often the case with large simulations or optimizations, identifying mistakes in the formulation is difficult. For instance, an incorrectly formulated constraint in an analysis model or connection template can result in the design space including no feasible

solution. Identifying these spurious constructs can be difficult in formulations with thousands of variables and constraints.

In addition, since a transformation approach is being used to transform between the information model-based representation of the problem and AIMMS code, there is the potential that this transformation process introduces unidentified errors. On the other hand, employing the transformation approach means that the formulation can be checked in the information models. There are a number of approaches for identifying errors in SysML and UML models (Alawneh, 2006), which could also be extended to the modeling approach presented in Chapter 3.

Also, the solvers are treated largely as black-box during this investigation. There are a number of tuning parameters and other mathematical programming specific tricks that could potentially change how a solver performs for a given problem, but these require experience in the mathematical programming domain.

8.4 Practical Implementation

It is important to consider how this approach could be implemented in the current environment found in industry. The major concern is whether existing team members (design engineers, test engineers, domain experts, and so on) have sufficient knowledge and expertise to make implementation of the presented framework feasible. Another issue is that as currently described, the framework has only an implicitly specified workflow. The transformation approach implies that users are first defining the exploration problem in SysML and then using the presented framework to transform that SysML model into a MIP optimization. The workflow for defining the SysML model is also only implicitly presented. In part, this is because it is likely that the workflow will need to be tailored to

the company implementing this framework. One potential workflow which was used to generate the models in this investigation was to begin with the definition of requirements, use the requirements to define the space of potential solutions, and then use both of those in the creation of the tests. In a commercial setting, it is likely that these tasks would be distributed to a team of systems engineers. The project workflow would also depend on the availability of model libraries. If all of the components being included in the system are already a part of the model library, then systems engineers can include those components in the formulation. If the components are not present, then domain experts need to be engaged as part of the process to help develop models (both structural and analytical) for the components that are not available in a model library. When comparing to other analysis tools that rely on model libraries to reduce the modeling effort, it is common for these model libraries to be created by the original tool vendor (such is the case with Simulink), a community of users and commercial companies (such as the case with Modelica) or by companies using the tool.

8.5 Open Questions and Opportunities for Future Research

8.5.1 Practical Aspects of Modeling an Architecture Selection Decision

One of the goals of this work is in establishing the potential value of explicitly modeling an architecture selection decision using information models. Realizing this potential value requires address several theoretical and practical questions:

What is the appropriate language for modeling an architecture selection decision? In this investigation, SysML was used to represent the architecture selection decision because of its availability and relatively universal acceptance within the MBSE community. SysML is a still evolving language with foundations in software design and

engineering, and it may not be the most suitable language for supporting system design during early phases of the design process. More investigation is needed on the tradeoffs between using SysML, adding the additional constructs proposed here as a normative or non-normative extension to SysML, or simply creating a language specifically tailored for representing these sorts of problems.

How can modeling tools be improved to support the modeling process? Current model authoring tools for SysML and similar languages are usually very generic. This has the advantage of allowing these tools to be used in a wide variety of different modeling methodologies in a wide range of domains. On the other hand, it makes constructing information models such as those presented in Chapters 3,4, and 7 more difficult.

How can the model best be visualized or represented to allow review by a number of diverse stakeholders? As mentioned earlier, one of the issues with using information models is accessibility to a wide-variety of stakeholders. One way to address this issue is through a transformation based approach where the transformations result in domain-specific artifacts that can be easily understood by the relevant stakeholder(s). The issue with a transformation approach is that separate transformations are needed for each artifact. A more potentially tractable solution would be to create a generic view-based partitioning approach where the aspects of the model could be quickly highlighted or removed as necessary.

How should emerging technology be included in the formulation? In Chapter 1, it was argued that design processes in domains with an influx of new technology have the potential to be significantly improved by employing expansive architecture

exploration processes. On the other hand, one way presented to reduce the modeling cost is to lean heavily on model reuse.

8.5.2 Practical Aspects of Solving an Architecture Selection Decision

Is mathematical programming the best path forward? In this work, mathematical programming solvers are shown to be capable of performing architecture exploration. Also, judging by the size of the problems considered, there is support for the hypothesis that they are more capable than the current state-of-the-art in architecture exploration or computational design synthesis. That being said, a more thorough comparison is needed.

What is the appropriate mathematical programming representation? In this work, the elements of the architecture selection decision were mapped to the mathematical programming domain, specifically mixed-integer linear programming, but only one potential representation was considered. One advantage of the transformation approach is that it can be easily modified to generate different representations which could then be characterized based on the solution times and quality of the solution(s) generated.

How can scale of the problem be further managed? In this investigation, some simple tricks were demonstrated for managing the scale of the problem. In general, these reduced the size of the mathematical programming problems which raises the issue: what is the right approach to decomposing the exploration problem so that it is manageable but can also support consistent decision making?

What is the right information to extract from the solver? In this investigation, the solver was queried for final solutions to the mathematical programming formulation.

It would also be interesting for designers to have access to the internal search tree of the solver, for instance which constraints are usually active.

What is the best way to characterize and visualize the results? In this investigation, the results of the optimization were presented as a single optimization modeled in SysML. When considering a pool of solutions, the issue becomes how to visualize that space of solutions. Because only a few architectures are considered in current explorations, the best way to represent a large space of possible architectures is not considered.

8.5.3 Extensions

What other types of analyses can be generated using the transformation approach to support the exploration process? As presented, the transformation and composition approach is fairly generic. As demonstrated in Chapter 4 and 6, it can be used both dynamic Modelica simulations and the mathematical programming formulation. There is the potential to use the same composition and flattening code to generate a number of other analyses from the architecture selection decision, for instance cost models.

How effective is the approach when applied to other application domains? In the current investigation, the focus was on hydraulic systems. These fit well with the approach because in the hydraulic domain components are modular and they have well-defined interfaces. This makes hydraulic components easy to compose using the technique presented in this research. There are many other domains where components are modular, so it is likely that the approach would be similarly applicable. One

additional complication is that in many domains the interfaces between the components are more complex.

What about controllers and software? In the current investigation, the controller and necessary software is excluded. The assumption is that cost of designing the software or controller is largely consistent across different architectures. This is obviously not always the case, so the question is how the controller or software design can be modeled in this framework. The difficulty is that up to this point, the types of models considered are algebraic models. How would the discrete nature of software or a controller fit with these algebraic models?

8.5.4 Informing Designers

Although anecdotally mentioned throughout the investigation, the overall goal of informing designer decision making is not tackled directly. Although the presented approach is capable of generating a single or multiple architectures from the prescribed formulation, user studies need to be performed to understand the impact that these results can have on changing the decisions a designer would make while designing a system. There is a dearth of such studies in the current literature for a number of reasons, from the limitations of current tools when applied to real-world problems to skepticism and lack of acceptance of computational tools into a process that is often considered an art.

8.6 Summary

This research is an investigation into using information models and mathematical programming to support decision making about the appropriate system architecture by facilitating architecture exploration. An information modeling language was created to represent architecture exploration problems as architecture selection decisions. Although

current trends in systems engineering have pushed designers from representing their design artifacts (requirements, architecture descriptions, and so forth) as documents to representing them as models, this investigation has pushed that boundary by not documenting a particular architecture or why it was selected but instead explicitly modeling the domain knowledge needed model an architecture exploration problem and select an appropriate architecture. This knowledge includes which alternatives should be considered and how to analyze and evaluate them. This allows designers to more explicitly represent their architecture exploration problem, to reach consensus about the knowledge that is included, and then to apply computational tools to this representation to help them select the best architecture.

The computational tools considered come from the mathematical programming domain. Since the architecture exploration process is an optimization process, the question becomes what is the appropriate representation of an architecture selection decision as an optimization problem and then what are the appropriate solvers. Because of the scope and discrete nature of the architecture space along with the need for continuous variables to size of a particular architecture, the mathematical programming domain is chosen as the domain of investigation because of the presence of languages that allow the solver independent encoding of the optimization problem and also high-quality commercial solvers to perform the optimization. In the current state-of-the-art, mathematical programming tools are not used during the system design process because of the difficult to formulate the optimization problem. To simplify this formulation process, this investigation considers representing the different pieces of the optimization problem in a modular fashion that can be composed. Using this framework as a template,

an automated transformation process is defined to convert the information model representation into mathematical programming optimization problems. Then the high-quality commercial solvers can be used on this problem.

Although there are currently significant limitations as outlined in Section 8.3, the contributions from this research are significant in the context of informational modeling as applied to Systems Engineering, utilizing composition and modularity to simplify the evaluation of architectures for architecture exploration and computational design synthesis, and also employing mathematical programming to perform the architecture exploration.

Whether the approach is the best solution is still unclear, however the contributions made in this research are useful points along the path toward an effective solution to the problem.

APPENDIX A:

MODEL LIBRARIES

In this appendix, the component models for the individual components, the analysis models used for the individual components, and the connection templates are presented.

A.1 Component-level structural models.

This section contains the commercial off-the-shelf components included in the component library. In this section, they are presented in tabular form; in the SysML model library each row corresponds to a separate Block with properties which are given default values. These models were automatically generated by importing the information represented in the table from files that contained the comma separated data. The information for the pumps and cylinders is based on previous component data available from Malak (Malak, 2008). The pumps considered are shown in Table A.1. The cylinders are shown in Table A.1. The data for the engines was synthesized specifically for the examples because accurate information about the brake specific fuel consumption is difficult to find for commercially available engines. The engines are shown in Table A.3. The interpolation values used to estimate the maximum torque are shown in Table A.4. The interpolation values for the fuel are shown in Table A.5.

Table A.1: Commercial off-the-shelf pumps.

ID	Displacement (m^3)	Max Pressure (Pa)	Max Speed (RPM)	Mass (kg)	Cost ($)
CPB-020	3.29E-05	24821126.24	3200	8.7496	837.8
CPB-023	3.67E-05	24821126.24	3200	8.8902	843.95
CPB-026	4.16E-05	24821126.24	3200	9.0718	850.21
CPB-030	4.79E-05	24821126.24	3200	9.2986	858.56
CPB-032	5.15E-05	24821126.24	3200	9.4803	866.67
CPB-035	5.57E-05	24821126.24	3200	9.6611	871.34
CPB-040	6.36E-05	24821126.24	3200	10.0661	883.77
CPB-045	7.16E-05	24821126.24	3000	10.4375	897.61
CPB-050	7.95E-05	22752699.06	2750	10.1688	907.37
CPB-055	8.78E-05	20684271.87	2500	11.5001	921.06
CPB-060	9.57E-05	18615844.68	2500	11.7315	934.97

Table A.2: Commercial off-the-shelf cylinders.

ID	Mass (kg)	Cost ($)	Bore Area (cm^2)	Rod Area (cm^2)
HMW-5008	32.60422	158.75	0.012668	0.0104
HMW-4008	18.50203	96.53	0.008107	0.006656
HMW-3508	14.80072	81.63	0.006207	0.005096
HMW-3008	11.20373	69.31	0.00456	0.003744
HMW-2508	9.003809	61.96	0.003167	0.0026
HMW-2008	6.100817	57	0.002027	0.001664
HMW-1508	5.202704	61.39	0.00114	0.000936

Table A.3: Commercial off-the-shelf engines.

ID	Mass (kg)	Cost ($)
E1	19.05	1200
E2	30.14	2000
E3	30.39	2000

Table A.4: Maximum torque (Nm) for a given normalized speed for the engines.

Normalized Speed	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
E1	9.4	13.7	17.3	20.3	22.7	24.4	25.6	26.0	25.9	25.1	23.7
E2	33.3	36.1	38.3	39.9	41.0	41.5	41.4	40.8	39.6	37.9	35.6
E3	24.3	29.8	34.4	38.1	40.9	42.9	44.0	44.2	43.5	42.0	39.6

Table A.5: Fuel consumption (kg/W) for a given normalized speed for the engines.

Normalized Speed	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
E1	0.5	0.5	0.5	0.5	0.49	0.45	0.49	0.5	0.5	0.5	0.5
E2	0.5	0.5	0.5	0.5	0.49	0.45	0.49	0.5	0.5	0.5	0.5
E3	0.5	0.5	0.5	0.5	0.49	0.45	0.49	0.5	0.5	0.5	0.5

A.2 Component-level analysis models.

In Figure A.1, the analysis model for the cylinder is illustrated. The cylinder has four interfaces (hydraulic ports *a* and *b* of type *Hydraulic* and translational ports *out* and *fixed* of type *Translational*), four variables (*boreArea*, the effective area on the bore side of the cylinder; *rodArea*, the effective area on the rod side of the cylinder; *force*, the force generated by the cylinder, and *velocity*, the velocity of the cylinder) , and three constraints. Each variable is stereotyped with «*MPVariable*», the *force* and *velocity* are tagged as variable (i.e., they change with the use scenarios) whereas *boreArea* and *rodArea* are tagged as parameters (i.e., they change with a particular design but not over the scenarios. The constraints are based on a force and flow balance at the piston. These constraints can be stated as:

$$\forall s\left(p_{a,s} \cdot A_b - p_{b,s} \cdot A_r + f_{out,s} = 0\right)$$

$$\forall s(Q_{a,s} \cdot A_b - Q_{b,s} \cdot A_r = 0)$$

$$\forall s(Q_{a,s} = v_{out,s} \cdot A_b)$$

where s is the set of use scenarios, $p_{a,s}$ and $p_{b,s}$ are the pressures at *a* and *b*, $Q_{a,s}$ and $Q_{b,s}$are the flows at *a* and *b*, $A_r$and $A_b$are the *rodArea* and *boreArea*, $f_{out,s}$ is the output force, and $v_{out,s}$is the output velocity.

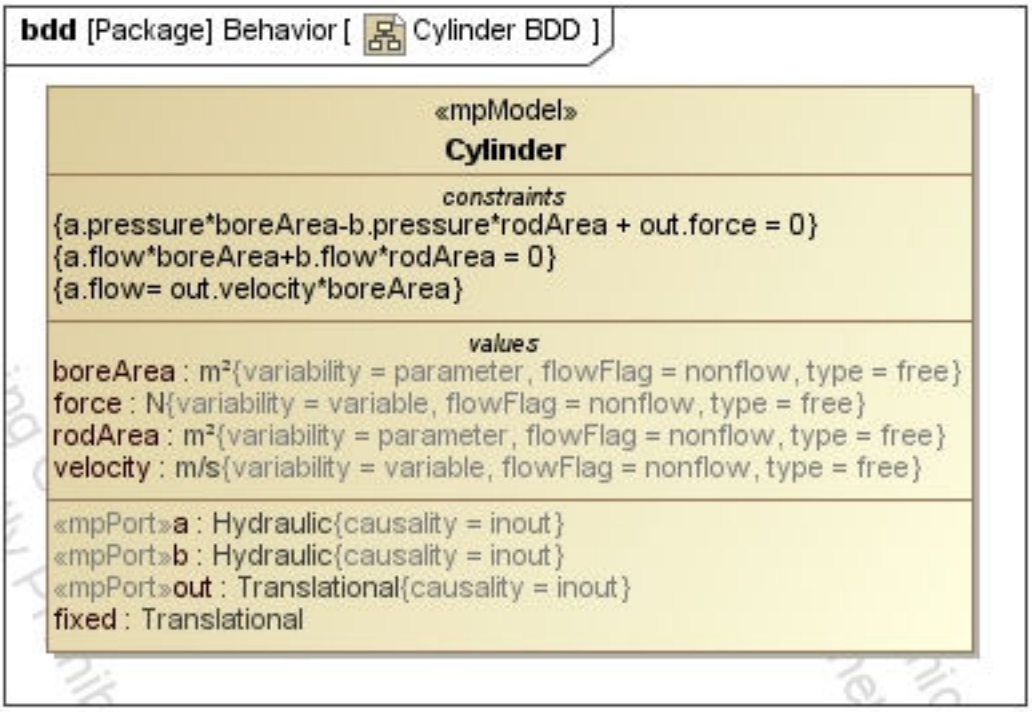

Figure A.1: Cylinder analysis model.

The analysis model for the engine is illustrated in Figure A.2. This model is a crude approximation of engine performance. It relies mostly on interpolating engine torque curves and brake specific fuel consumption maps. The maximum torque the engine produces is considered to be a function of the engine speed.

Figure A.2: Engine analysis model

A schematic for the open center valve is shown in Figure A.3; this valve is an open-center neutral pass-through valve. Unlike a traditional open-center valve, it has six instead of four interfaces to accommodate the neutral pass-through. The valve has three operating modes, which have been labeled *on*, *off*, and *back*.

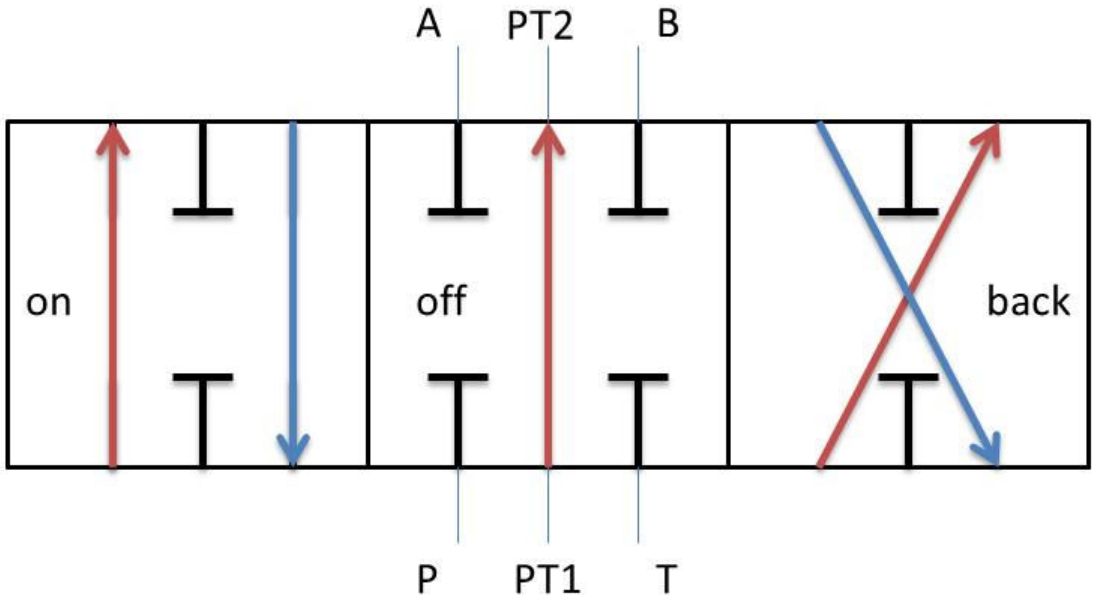

Figure A.3: Illustration of the valve's schematic. The valve has three operating modes. Each mode is in a different quadrant. Also, the red arrows indicate high pressure flow and blue arrows indicate low pressure flow.

The analysis model for the valve is shown in Figure A.4, valve losses have been neglected and the emphasis has been placed on insuring that Kirchhoff's laws are correctly enforced during each operating mode. This is done by using conditional indicator constraints that are active during only the appropriate operational mode. Equations describing the general flow are always active:

$$\forall s(Q_{P,s} + Q_{PA,s} + Q_{PB,s} = 0)$$

$$\forall s(Q_{T,s} + Q_{TA,s} + Q_{TB,s} = 0)$$

$$\forall s(Q_{A,s} + Q_{PA,s} + Q_{TA,s} = 0)$$

$$\forall s(Q_{B,s} + Q_{PB,s} + Q_{TB,s} = 0)$$

$$\forall s(Q_{PT1,s} + Q_{PT2,s} = 0)$$

where $Q_{P,s}$, $Q_{T,s}$, $Q_{A,s}$, $Q_{B,s}$,$Q_{PT1,s}$, $Q_{PT2,s}$ are the flows at ports *P*, *T*, *A*, *B*, *PT1*, and *PT2* respectively. $Q_{PA,s}$, $Q_{TA,s}$, $Q_{PB,s}$, $Q_{TB,s}$are the flows between ports P and A; Ports T and A; ports P and B; and ports T and B, respectively.

For the off operating mode, the following flow equations are specified:

$$\forall s(Q_{PA,s} = 0)$$

$$\forall s(Q_{PB,s} = 0)$$

$$\forall s(Q_{TA,s} = 0)$$

$$\forall s(Q_{TB,s} = 0)$$

assuring that flow is only possible through the neutral pass-through. For the on operating mode, the following flow equations are specified:

$$\forall s(Q_{PT1,s} = 0)$$

$$\forall s(Q_{PB,s} = 0)$$

$$\forall s(Q_{TA,s} = 0)$$

assuring that flow is only possible between *P* and *A*, and *T* and *B*. For the back operating mode, the following equations are specified:

$$\forall s(Q_{PT1,s} = 0)$$

$$\forall s(Q_{PA,s} = 0)$$

$$\forall s(Q_{TB,s} = 0)$$

Assuring that flow is only possible in the other direction. A similar set of equations is used for the pressures; when there is potential for flow between two ports the pressures of these ports are equated. This neglects pressure losses across the valve (which is a major source of energy loss in real systems).

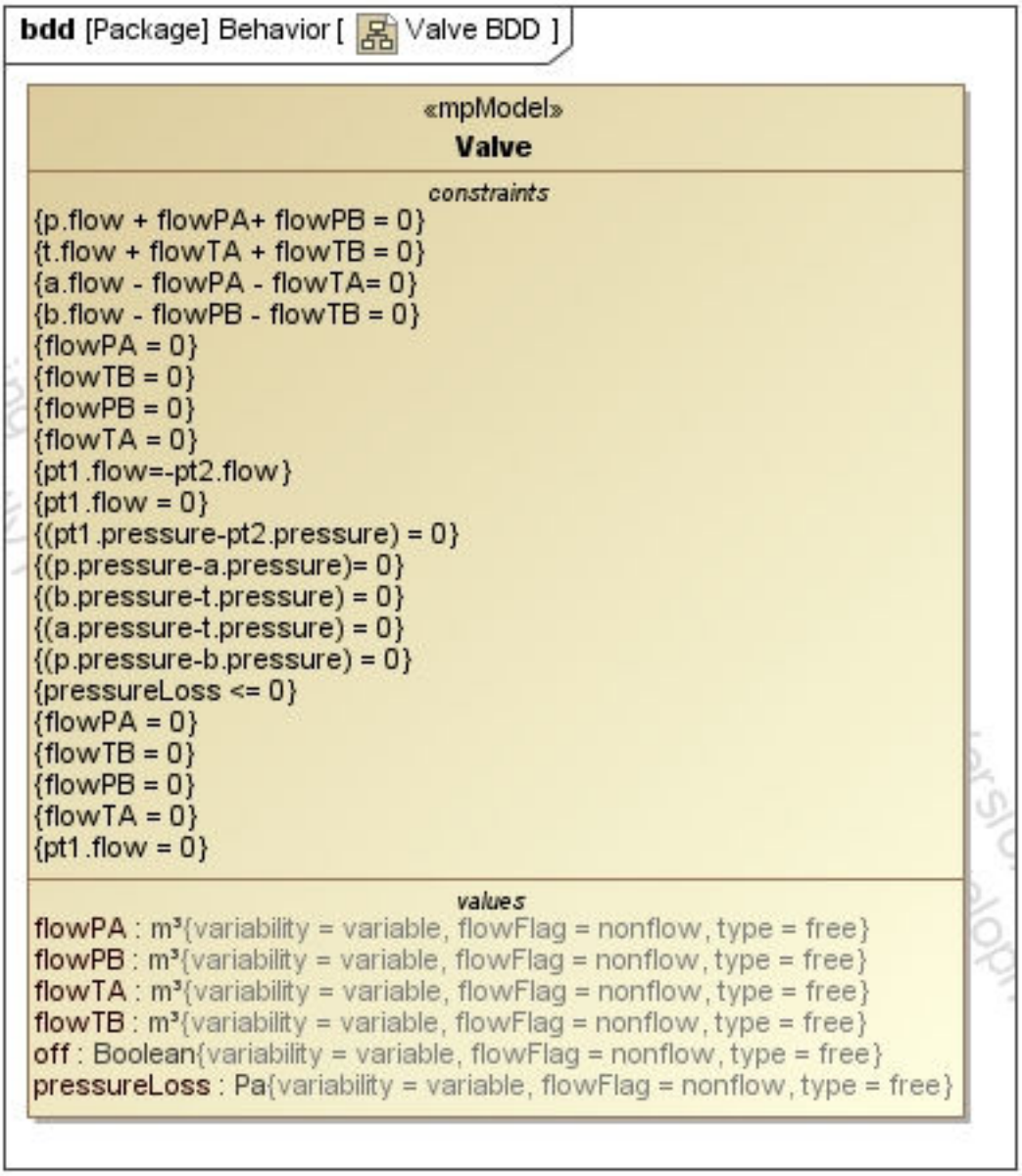

Figure A.4: Open center valve with neutral pass through analysis model.

The schematic for the closed center valve is shown in Figure A.5. This schematic is similar to the open center neutral pass-through valve, but lacks a neutral pass-through. The analysis model for the closed-center valve considered is the same as the analysis model for the open-center valve, except the interfaces and constraints for the pass-through are removed. This analysis model is shown in Figure A.6.

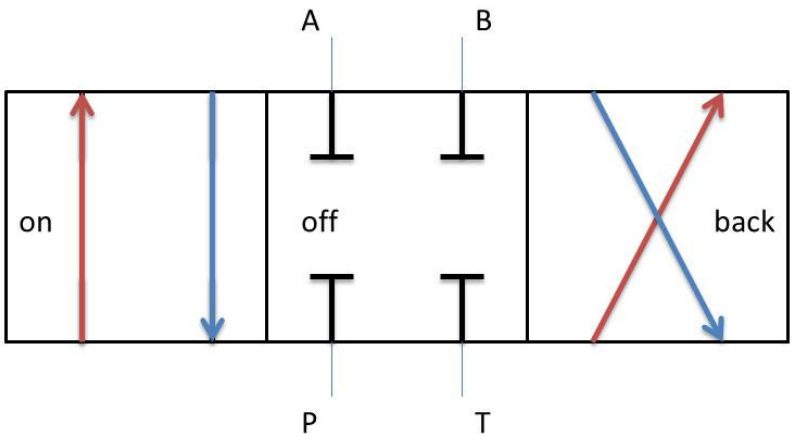

Figure A.5: Schematic for the closed center valve.

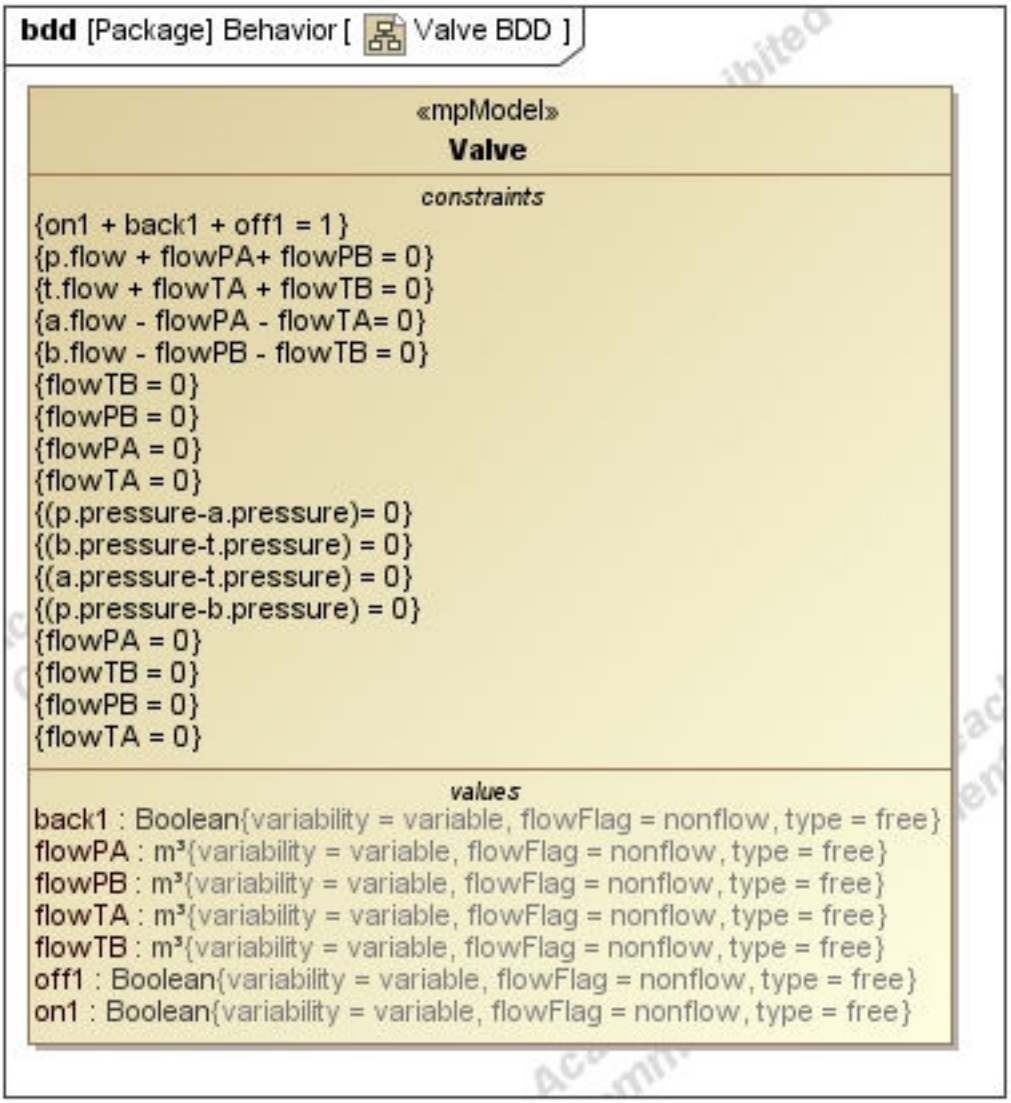

Figure A.6: Analysis model for the closed center valve.

The analysis model for the pump is shown in Figure A.7. This analysis model only captures the pumps behavior in the forward pumping phase. It is based on the McCandlish pump equations(McCandlish, 1984). The pump model has three interfaces which are inherited from a generic model of a pump, hydraulic interfaces *p1* and *t1* for fluid to flow into and out of the pump and rotational interface *in1* where the pump can be connected to a prime mover (such as an engine or motor). The pump also has the *disp* (*D*)

parameter which describes the size of the pump, along with a number of parameters that are used to calculate the losses in the pump. These parameters are as follows: *B* is the fluid bulk modulus, *cf* (C_f) is the Coulomb friction coefficient, *cs* is the slip coefficient (C_s), *cv* (C_v) is the viscous friction coefficient, *Vr* (V_r) is the volume ratio, and *mu* (μ) is the fluid absolute viscosity. These are also parameters that depend on the particular pump (and fluid) selected. The rest of the variables are intermediate variables used during the determination of pump behavior. The pump constraints can be expressed as follows:

$$Q_{p1,s} = D \cdot \left[\omega_{in1,s} - C_s\left(\frac{p_{p1,s}}{\mu}\right) - \left(\frac{\omega_{in1,s} \cdot p_{p1,s}}{B}\right) \cdot (V_r + 1)\right]$$

$$\mathrm{T}_{in1,s} = D \cdot \left[p_{p1,s} + C_v\left(\mu \cdot \omega_{in1,s}\right) + C_f \cdot p_{p1,s}\right]$$

with the assumption that the load dependent friction is independent of the displacement. For the most part, these constraints contain parameters multiplied by variables and therefore do not need to be approximated with interpolation. On the other hand, the multiplication of $\omega_{in1,s} \cdot p_{p1,s}$ is approximated using the interpolation described in Section 5.4.3.

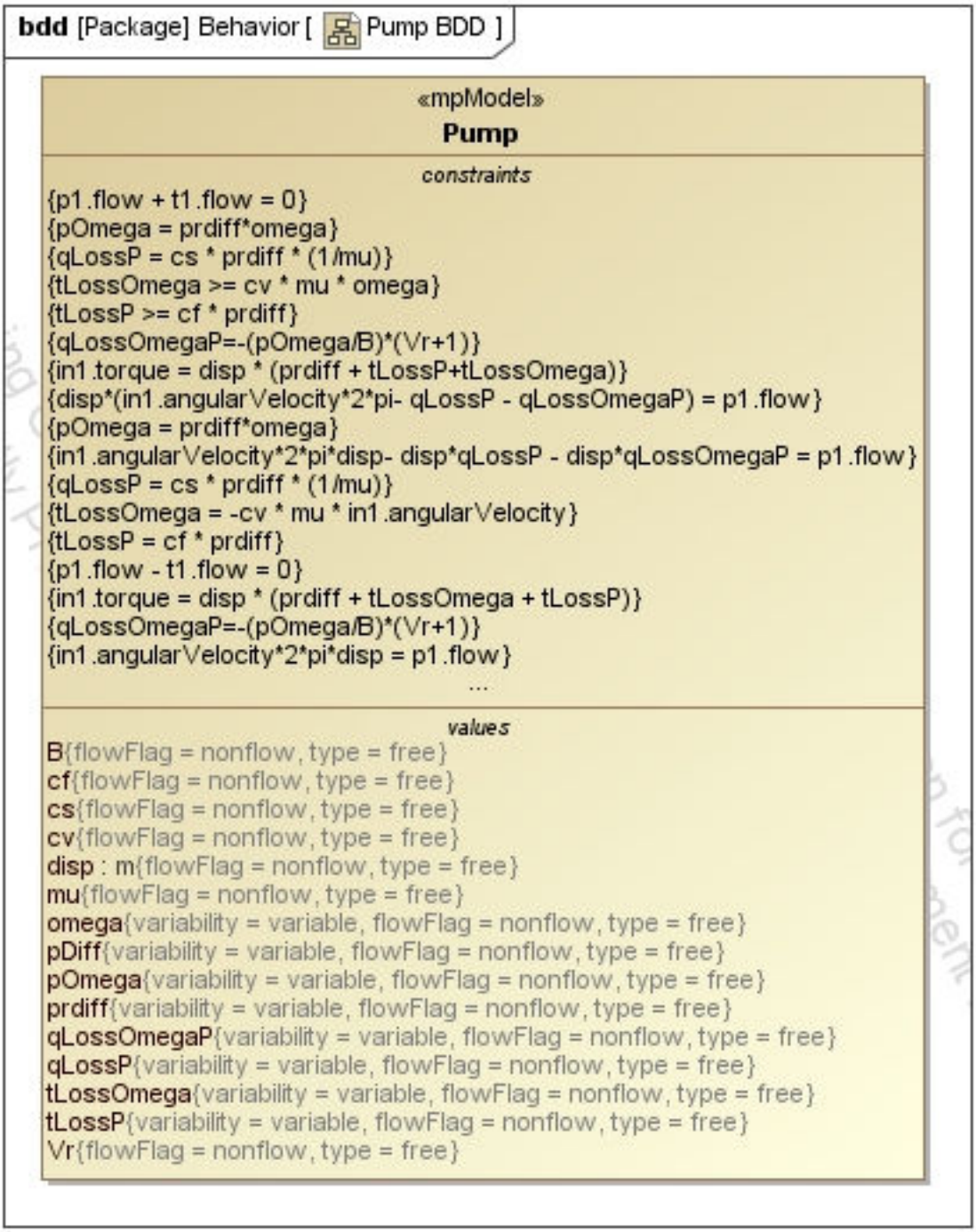

Figure A.7: Fixed-displacement pump analysis model.

A.3 Functional Units

In this section, the functional units that are included in the functional units library are presented. These functional units include components which are commonly combined. A fixed-displacement power unit (this functional unit is often referred to as an open center power unit because it is usually combined with open center valves) is shown in Figure A.8, this functional unit includes a fixed-displacement pump, a tank, and a relief valve across the pump to insure that the pump side pressure does not exceed the maximum pressure. An open center valve block is shown in Figure A.9; this valve block includes an open center valve which has been paired with a check valve to prevent cavitation. The same is true for the closed center valve block shown in Figure A.10; it contains a closed-center valve and a check valve to prevent cavitation.

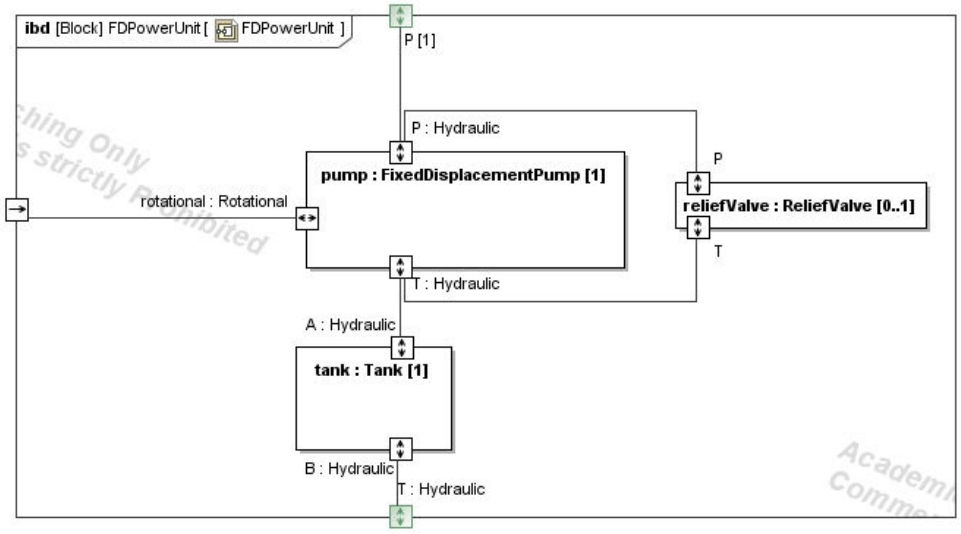

Figure A.8: Fixed-displacement power unit.

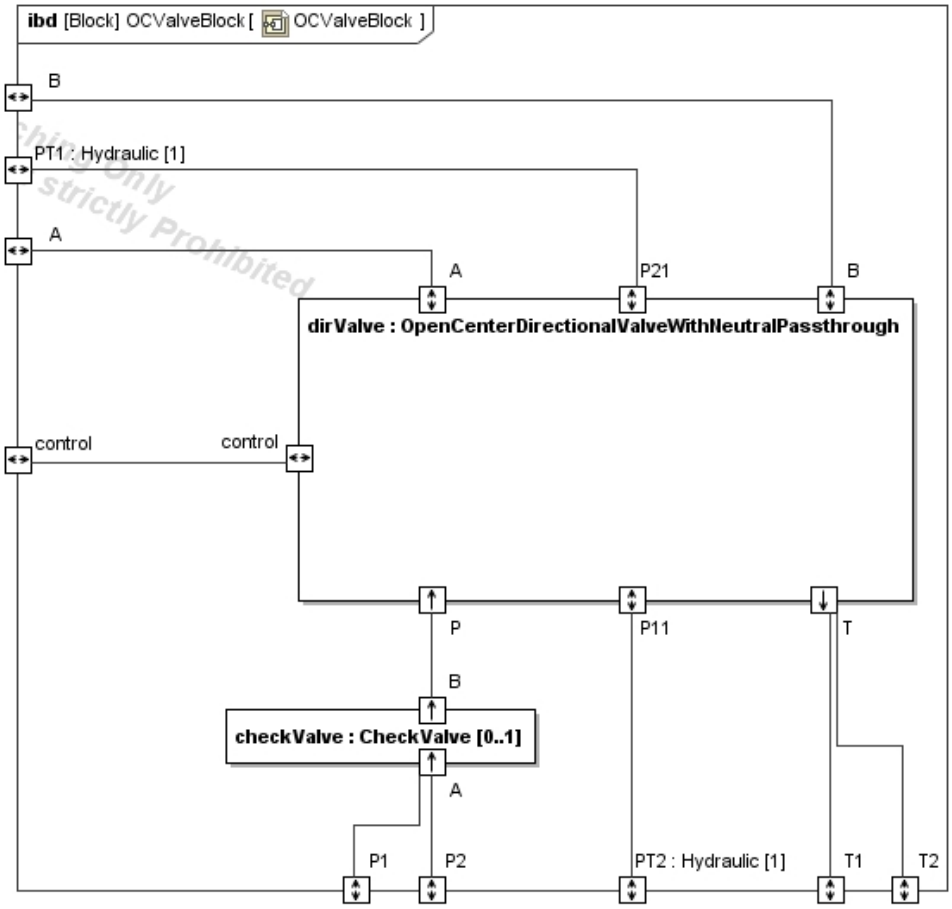

Figure A.9: Open center valve block.

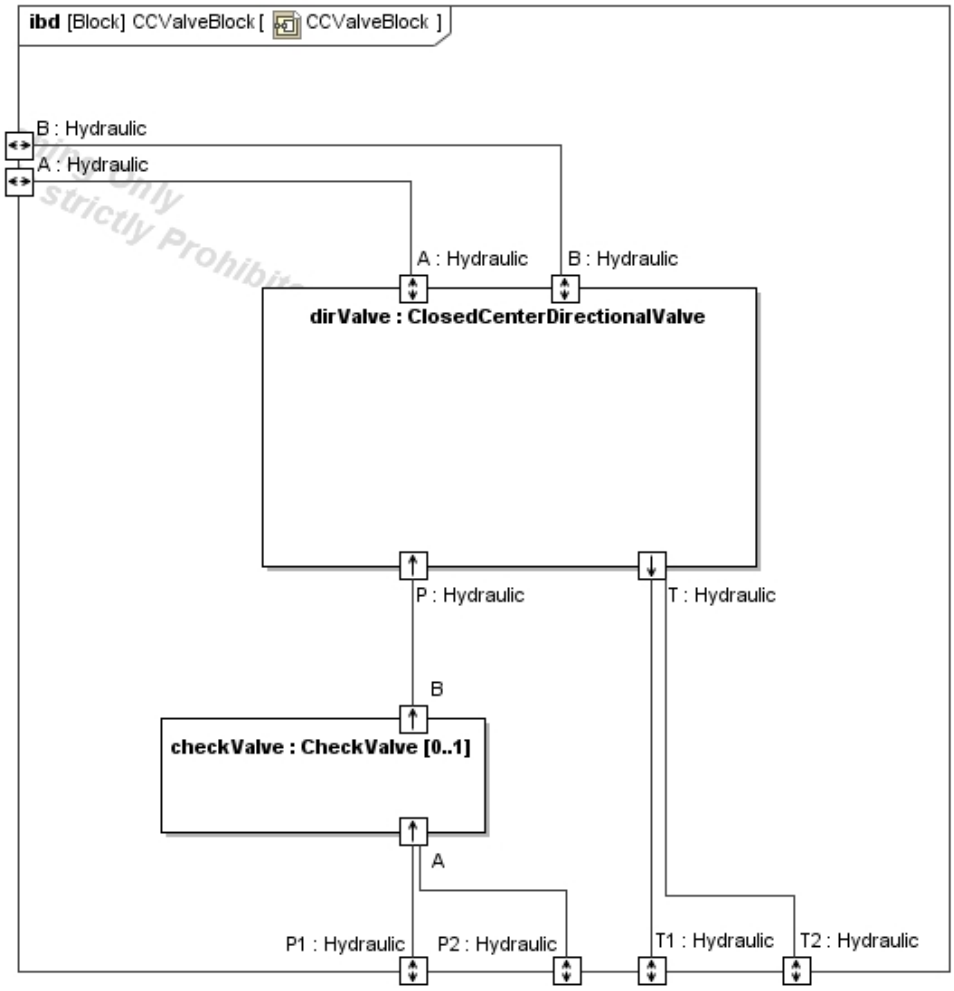

Figure A.10: Closed center valve block.

A.4 Connection Templates

In this section, the connection templates that are included are presented. These connection templates describe how the components are traditionally connected together. The different connection templates that are included are summarized in Figure A.11.

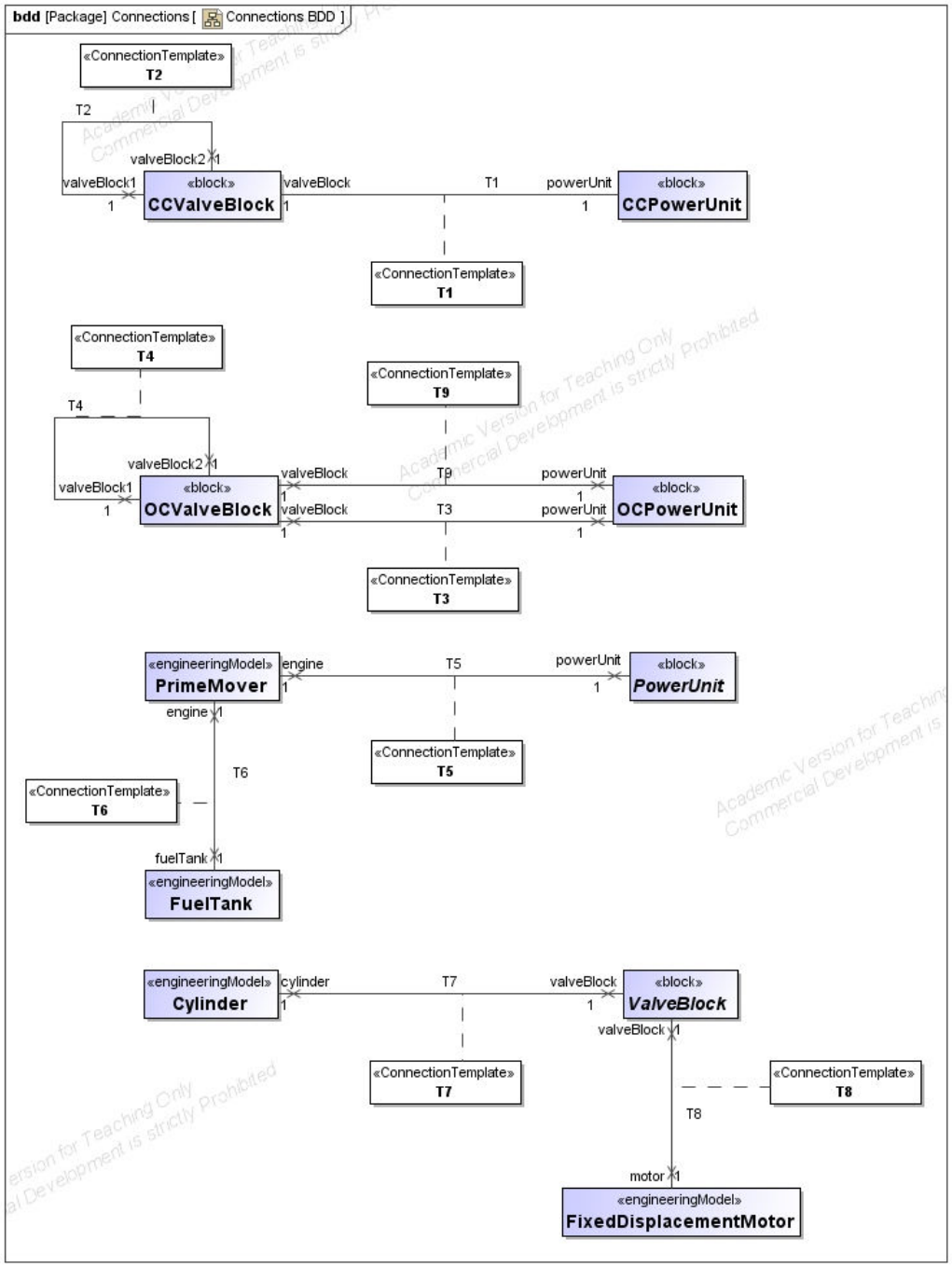

Figure A.11: Connection templates represented in the component library

Figure A.12, Figure A.13, and Figure A.14 describe connections between specific functional units. For instance, the connection between an open center power unit and an open center valve block. On the other hand, Figure A.15, Figure A.16, and Figure A.17 describe more generic connections. Instead of worrying about the type of power unit (open center, closed center, etc.), the template between a prime mover (engine) and power unit is done at a generic level. The assumption is that any specialization of these generic components is then connected in the same way. Therefore, the designer does not need to define a

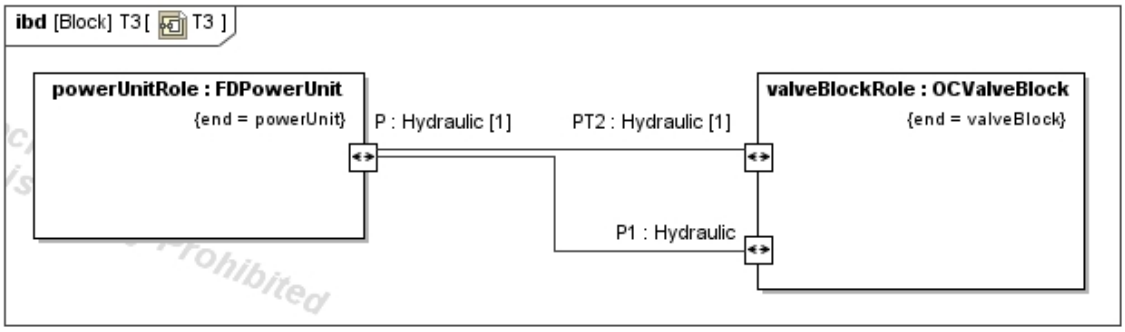

Figure A.12: Connection template between a fixed-displacement power unit and an open-center valve block.

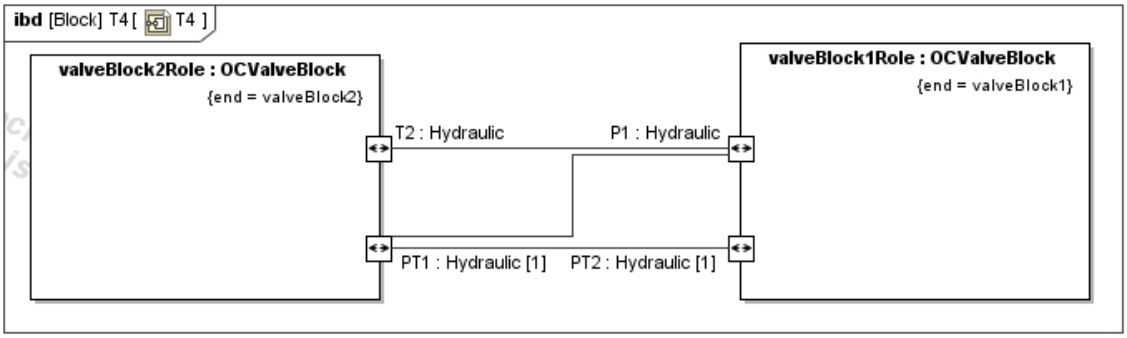

Figure A.13: Connection template between an open-centered valve block and another open-center valve block.

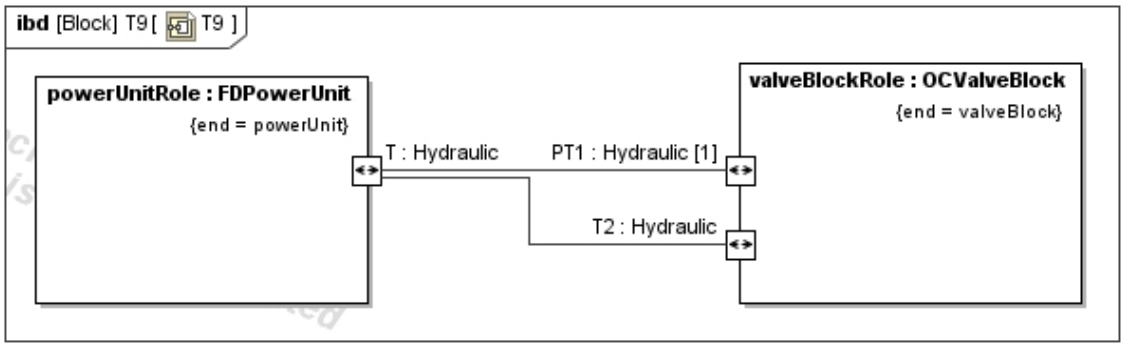

Figure A.14: Connection between a fixed-displacement power unit and an open-centered valve block

.

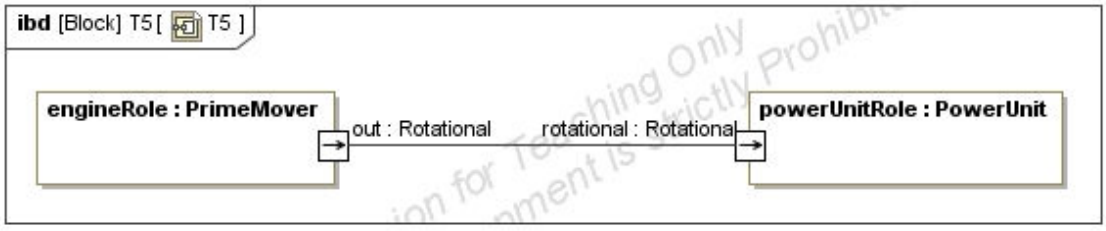

Figure A.15: Connection between a prime mover (engine) and a (generic) power unit.

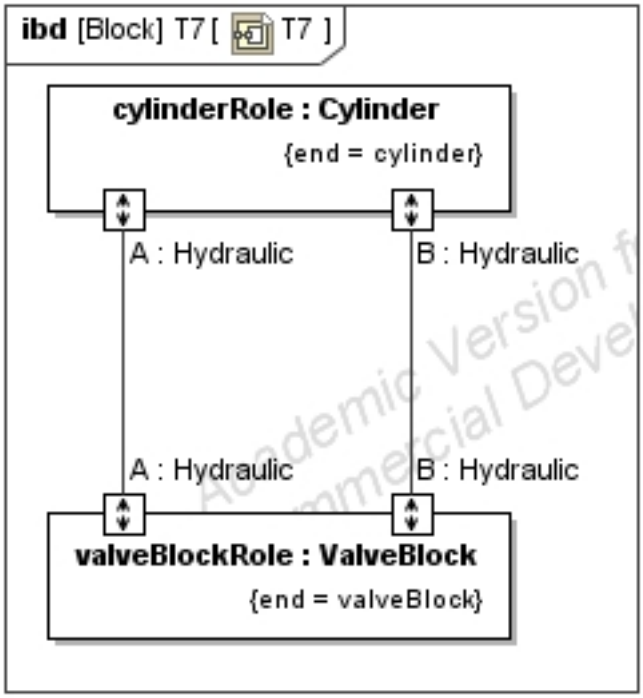

Figure A.16: Connection between a cylinder and a (generic) valve block.

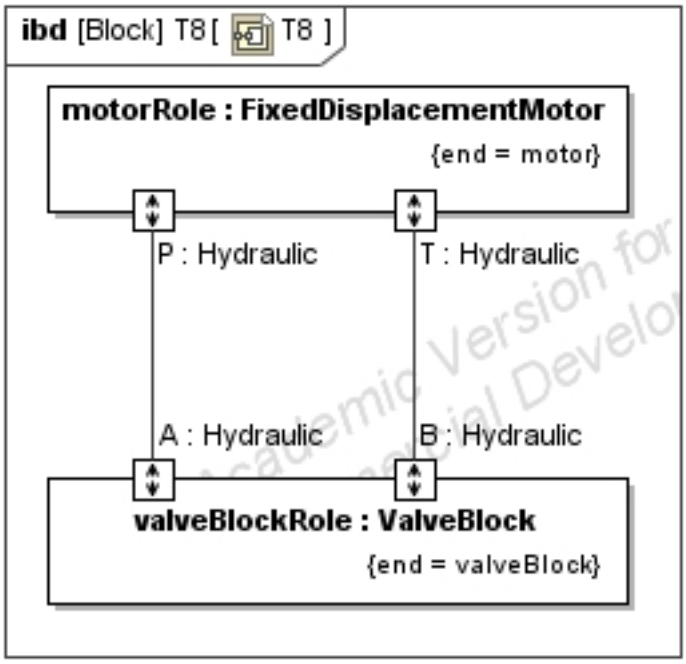

Figure A.17: Connection between a motor and a (generic) valve block.

APPENDIX B:

SAMPLE AIMMS CODE

```
1.  MAIN MODEL View
2.  DECLARATION SECTION
3.  PARAMETER:
4.  identifier : pi
5.  definition : 4*arctan(1);
6.  SET:
7.  identifier : states
8.  indices : s
9.  definition : data {'1','2' };
10. SET:
11. identifier : cots
12. indices : cotsIndex
13. definition : data {'1','2','3','4','5','6','7','8','9','10','11','12','13','14','15','16','17','18','19' };
14. SET:
15. identifier : components
16. indices : c
17. definition : data {'1','2','3','4','5','6','7','8','9','10' };
18. VARIABLE:
19. identifier : cotsVar
20. index domain : cotsIndex
21. range : binary;
22. PARAMETER:
23. identifier : cotsWeights
24. index domain : cotsIndex
25. definition : data {1 : 1,2 : 2,3 : 3,4 : 4,5 : 5,6 : 6,7 : 7,8 : 8,9 : 9,10 : 10,11 : 11,12 : 12,13 :
    13,14 : 14,15 : 15,16 : 16,17 : 17,18 : 18,19 : 19};
26. PARAMETER:
27. identifier : cotsBound
28. definition : 1e6;
29. CONSTRAINT:
30. identifier : s0
31. property : SOS1
32. sos weight : cotsVar(cotsIndex) : cotsWeights(cotsIndex)
33. definition : -1+ cotsVar('1')+ cotsVar('2')+ cotsVar('3')+ cotsVar('4')+ cotsVar('5')+
    cotsVar('6')+ cotsVar('7')+ cotsVar('8')+ cotsVar('9')+ cotsVar('10')+ cotsVar('11') = 0;
34. CONSTRAINT:
35. identifier : s1
36. property : SOS1
37. sos weight : cotsVar(cotsIndex) : cotsWeights(cotsIndex)
38. definition : -1+ cotsVar('12') = 0;
39. CONSTRAINT:
40. identifier : s2
41. property : SOS1
42. sos weight : cotsVar(cotsIndex) : cotsWeights(cotsIndex)
43. definition : -1+ cotsVar('13')+ cotsVar('14')+ cotsVar('15')+ cotsVar('16')+ cotsVar('17')+
    cotsVar('18')+ cotsVar('19') = 0;
44. SET:
45. identifier : ports
46. indices : p
```

```
47.   definition : data
      {'1','2','3','4','5','6','7','8','9','10','11','12','13','14','15','16','17','18','19','20','21','22','23','24' };
48.   SET:
49.   identifier : translational
50.   indices : t
51.   definition : data {'19','20','23','24' };
52.   SET:
53.   identifier : rotational
54.   indices : r
55.   definition : data {'1','16' };
56.   SET:
57.   identifier : hydraulic
58.   indices : h
59.   definition : data {'2','3','4','5','6','7','8','9','10','11','12','13','14','15','17','18','21','22' };
60.   SET:
61.   identifier : decisions
62.   indices : d
63.   definition : data {'1','2','3','4','5','6','7' };
64.   /*c0:
65.   1=>disp
66.   2=>qLossOmegaP
67.   3=>pDiff
68.   4=>B
69.   5=>tLossP
70.   6=>qLossP
71.   7=>cs
72.   8=>pOmega
73.   9=>Vr
74.   10=>cv
75.   11=>mu
76.   12=>omega
77.   13=>prdiff
78.   14=>tLossOmega
79.   15=>cf
80.   16=>displacement
81.   17=>maxOpSpeed
82.   18=>maxOpPr
83.   19=>cost
84.   20=>mass
85.   c1:
86.   c2:
87.   c3:
88.   21=>back
89.   22=>on
90.   23=>ctrl
91.   24=>off
92.   25=>flowPA
93.   26=>flowPB
94.   27=>flowTA
95.   28=>flowTB
96.   29=>pressureLoss
97.   30=>cost
98.   31=>mass
99.   32=>maxFlow
100.  33=>maxPr
101.  c4:
```

```
c5:
34=>power
35=>tau
36=>omega
37=>maxTau
38=>normalizedSpeed
39=>fuelConsumption
40=>fuelRate
41=>maxSpeed
42=>minSpeed
43=>minTau
44=>cost
45=>mass
46=>rpmMin
47=>rpmMax
48=>a0
49=>a1
50=>a2
51=>thermalEff
c6:
52=>force
53=>velocity
54=>boreArea
55=>rodArea
56=>boreDiameter
57=>strokeLength
58=>mass
59=>cost
60=>maxPressure
61=>rodDiameter
62=>v0
63=>v1
64=>v2
65=>v3
66=>v4
c7:
c8:
c9:
67=>force
68=>velocity
c0:
1=>in1
2=>p1
3=>t1
c1:
4=>in
5=>out
c2:
6=>high
7=>low
c3:
8=>p
9=>t
10=>a
11=>b
12=>pt1
```

```
13=>pt2
c4:
14=>in
15=>out
c5:
16=>out1
c6:
17=>a
18=>b
19=>out
20=>fixed
c7:
21=>in
22=>out
c8:
23=>flange
c9:
24=>flange
1=>reliefvalve0
2=>checkvalve0
3=>T3.1
4=>T9.1
5=>T5.1.1
6=>T4.1
7=>T7.1.1
*/
VARIABLE:
identifier : force
index domain : (t, s)
range : free;
PARAMETER:
identifier : forceBound
definition : 1e9;
VARIABLE:
identifier : velocity
index domain : (t, s)
range : free;
PARAMETER:
identifier : velocityBound
definition : 1e9;
VARIABLE:
identifier : angularVelocity
index domain : (r, s)
range : free;
PARAMETER:
identifier : angularVelocityBound
definition : 1e9;
VARIABLE:
identifier : torque
index domain : (r, s)
range : free;
PARAMETER:
identifier : torqueBound
definition : 1e9;
VARIABLE:
identifier : pressure
```

```
214.  index domain : (h, s)
215.  range : nonnegative;
216.  PARAMETER:
217.  identifier : pressureBound
218.  definition : 1e9;
219.  VARIABLE:
220.  identifier : flow
221.  index domain : (h, s)
222.  range : free;
223.  PARAMETER:
224.  identifier : flowBound
225.  definition : 1e9;
226.  SET:
227.  identifier : ctrlVariableSet
228.  indices : ctrlVariableSetIndex
229.  definition : data {'23' };
230.  VARIABLE:
231.  identifier : ctrlVariable
232.  index domain : (ctrlVariableSetIndex,s)
233.  range : free;
234.  SET:
235.  identifier : qLossOmegaPVariableSet
236.  indices : qLossOmegaPVariableSetIndex
237.  definition : data {'2' };
238.  VARIABLE:
239.  identifier : qLossOmegaPVariable
240.  index domain : (qLossOmegaPVariableSetIndex,s)
241.  range : free;
242.  PARAMETER:
243.  identifier : massParameter
244.  definition : 1;
245.  SET:
246.  identifier : pressureLossVariableSet
247.  indices : pressureLossVariableSetIndex
248.  definition : data {'29' };
249.  VARIABLE:
250.  identifier : pressureLossVariable
251.  index domain : (pressureLossVariableSetIndex,s)
252.  range : free;
253.  SET:
254.  identifier : powerVariableSet
255.  indices : powerVariableSetIndex
256.  definition : data {'34' };
257.  VARIABLE:
258.  identifier : powerVariable
259.  index domain : (powerVariableSetIndex,s)
260.  range : free;
261.  SET:
262.  identifier : onVariableSet
263.  indices : onVariableSetIndex
264.  definition : data {'22' };
265.  VARIABLE:
266.  identifier : onVariable
267.  index domain : (onVariableSetIndex,s)
268.  range : binary;
269.  SET:
```

```
identifier : flowPAVariableSet
indices : flowPAVariableSetIndex
definition : data {'25' };
VARIABLE:
identifier : flowPAVariable
index domain : (flowPAVariableSetIndex,s)
range : free;
SET:
identifier : tLossOmegaVariableSet
indices : tLossOmegaVariableSetIndex
definition : data {'14' };
VARIABLE:
identifier : tLossOmegaVariable
index domain : (tLossOmegaVariableSetIndex,s)
range : free;
SET:
identifier : fuelRateVariableSet
indices : fuelRateVariableSetIndex
definition : data {'40' };
VARIABLE:
identifier : fuelRateVariable
index domain : (fuelRateVariableSetIndex,s)
range : free;
SET:
identifier : flowPBVariableSet
indices : flowPBVariableSetIndex
definition : data {'26' };
VARIABLE:
identifier : flowPBVariable
index domain : (flowPBVariableSetIndex,s)
range : free;
PARAMETER:
identifier : maxFlowParameter
definition : 1;
PARAMETER:
identifier : minTauParameter
definition : 1;
SET:
identifier : normalizedSpeedVariableSet
indices : normalizedSpeedVariableSetIndex
definition : data {'38' };
VARIABLE:
identifier : normalizedSpeedVariable
index domain : (normalizedSpeedVariableSetIndex,s)
range : free;
PARAMETER:
identifier : thermalEffParameter
definition : 1;
SET:
identifier : cvParameterSet
indices : cvParameterSetIndex
definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
PARAMETER:
identifier : cvParameter
index domain : cvParameterSetIndex
```

```
definition : data {1 : 2E3,2 : 2E3,3 : 2E3,4 : 2E3,5 : 2E3,6 : 2E3,7 : 2E3,8 : 2E3,9 : 2E3,10 : 2E3,11: 2E3 };
PARAMETER:
identifier : costParameter
definition : 1;
SET:
identifier : omegaVariableSet
indices : omegaVariableSetIndex
definition : data {'12','36' };
VARIABLE:
identifier : omegaVariable
index domain : (omegaVariableSetIndex,s)
range : free;
SET:
identifier : csParameterSet
indices : csParameterSetIndex
definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
PARAMETER:
identifier : csParameter
index domain : csParameterSetIndex
definition : data {1 : 3E-9,2 : 3E-9,3 : 3E-9,4 : 3E-9,5 : 3E-9,6 : 3E-9,7 : 3E-9,8 : 3E-9,9 : 3E-9,10 : 3E-9,11: 3E-9 };
SET:
identifier : rodAreaParameterSet
indices : rodAreaParameterSetIndex
definition : data {'1','2','3','4','5','6','7' };
PARAMETER:
identifier : rodAreaParameter
index domain : rodAreaParameterSetIndex
definition : data {1 : 0.010399546,2 : 0.00665571,3 : 0.005095778,4 : 0.003743837,5 : 0.002599887,6 : 0.000935959,7: 0.001663927 };
SET:
identifier : maxSpeedParameterSet
indices : maxSpeedParameterSetIndex
definition : data {'1' };
PARAMETER:
identifier : maxSpeedParameter
index domain : maxSpeedParameterSetIndex
definition : data {1: 377 };
SET:
identifier : tauVariableSet
indices : tauVariableSetIndex
definition : data {'35' };
VARIABLE:
identifier : tauVariable
index domain : (tauVariableSetIndex,s)
range : free;
SET:
identifier : tLossPVariableSet
indices : tLossPVariableSetIndex
definition : data {'5' };
VARIABLE:
identifier : tLossPVariable
index domain : (tLossPVariableSetIndex,s)
range : free;
SET:
```

```
identifier : dispParameterSet
indices : dispParameterSetIndex
definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
PARAMETER:
identifier : dispParameter
index domain : dispParameterSetIndex
definition : data {1 : 3.29E-05,2 : 3.67E-05,3 : 4.16E-05,4 : 4.79E-05,5 : 5.15E-05,6 : 5.57E-
05,7 : 6.36E-05,8 : 7.16E-05,9 : 7.95E-05,10 : 8.78E-05,11: 9.57E-05 };
SET:
identifier : cfParameterSet
indices : cfParameterSetIndex
definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
PARAMETER:
identifier : cfParameter
index domain : cfParameterSetIndex
definition : data {1 : .094,2 : .094,3 : .094,4 : .094,5 : .094,6 : .094,7 : .094,8 : .094,9 : .094,10 :
.094,11: .094 };
SET:
identifier : qLossPVariableSet
indices : qLossPVariableSetIndex
definition : data {'6' };
VARIABLE:
identifier : qLossPVariable
index domain : (qLossPVariableSetIndex,s)
range : free;
SET:
identifier : pDiffVariableSet
indices : pDiffVariableSetIndex
definition : data {'3' };
VARIABLE:
identifier : pDiffVariable
index domain : (pDiffVariableSetIndex,s)
range : free;
PARAMETER:
identifier : maxOpSpeedParameter
definition : 1;
PARAMETER:
identifier : a0Parameter
definition : 1;
SET:
identifier : BParameterSet
indices : BParameterSetIndex
definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
PARAMETER:
identifier : BParameter
index domain : BParameterSetIndex
definition : data {1 : 1.66E9,2 : 1.66E9,3 : 1.66E9,4 : 1.66E9,5 : 1.66E9,6 : 1.66E9,7 :
1.66E9,8 : 1.66E9,9 : 1.66E9,10 : 1.66E9,11: 1.66E9 };
SET:
identifier : pOmegaVariableSet
indices : pOmegaVariableSetIndex
definition : data {'8' };
VARIABLE:
identifier : pOmegaVariable
index domain : (pOmegaVariableSetIndex,s)
range : free;
```

```
PARAMETER:
identifier : maxPressureParameter
definition : 1;
SET:
identifier : minSpeedParameterSet
indices : minSpeedParameterSetIndex
definition : data {'1' };
PARAMETER:
identifier : minSpeedParameter
index domain : minSpeedParameterSetIndex
definition : data {1: 104.2 };
PARAMETER:
identifier : displacementParameter
definition : 1;
PARAMETER:
identifier : boreDiameterParameter
definition : 1;
PARAMETER:
identifier : a1Parameter
definition : 1;
SET:
identifier : velocityVariableSet
indices : velocityVariableSetIndex
definition : data {'53','68' };
VARIABLE:
identifier : velocityVariable
index domain : (velocityVariableSetIndex,s)
range : free;
PARAMETER:
identifier : a2Parameter
definition : 1;
SET:
identifier : prdiffVariableSet
indices : prdiffVariableSetIndex
definition : data {'13' };
VARIABLE:
identifier : prdiffVariable
index domain : (prdiffVariableSetIndex,s)
range : free;
PARAMETER:
identifier : VrParameter
definition : 1;
SET:
identifier : backVariableSet
indices : backVariableSetIndex
definition : data {'21' };
VARIABLE:
identifier : backVariable
index domain : (backVariableSetIndex,s)
range : binary;
SET:
identifier : muParameterSet
indices : muParameterSetIndex
definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
PARAMETER:
identifier : muParameter
```

```
index domain : muParameterSetIndex
definition : data {1 : .017,2 : .017,3 : .017,4 : .017,5 : .017,6 : .017,7 : .017,8 : .017,9 : .017,10 : .017,11: .017 };
SET:
identifier : fuelConsumptionVariableSet
indices : fuelConsumptionVariableSetIndex
definition : data {'39' };
VARIABLE:
identifier : fuelConsumptionVariable
index domain : (fuelConsumptionVariableSetIndex,s)
range : free;
PARAMETER:
identifier : maxOpPrParameter
definition : 1;
SET:
identifier : maxTauVariableSet
indices : maxTauVariableSetIndex
definition : data {'37' };
VARIABLE:
identifier : maxTauVariable
index domain : (maxTauVariableSetIndex,s)
range : free;
PARAMETER:
identifier : maxPrParameter
definition : 1;
PARAMETER:
identifier : rodDiameterParameter
definition : 1;
SET:
identifier : offVariableSet
indices : offVariableSetIndex
definition : data {'24' };
VARIABLE:
identifier : offVariable
index domain : (offVariableSetIndex,s)
range : binary;
SET:
identifier : flowTBVariableSet
indices : flowTBVariableSetIndex
definition : data {'28' };
VARIABLE:
identifier : flowTBVariable
index domain : (flowTBVariableSetIndex,s)
range : free;
PARAMETER:
identifier : rpmMaxParameter
definition : 1;
SET:
identifier : boreAreaParameterSet
indices : boreAreaParameterSetIndex
definition : data {'1','2','3','4','5','6','7' };
PARAMETER:
identifier : boreAreaParameter
index domain : boreAreaParameterSetIndex
definition : data {1 : 0.012667687,2 : 0.00810732,3 : 0.006207167,4 : 0.004560367,5 : 0.003166922,6 : 0.001140092,7: 0.00202683 };
```

```
SET:
identifier : forceVariableSet
indices : forceVariableSetIndex
definition : data {'52','67' };
VARIABLE:
identifier : forceVariable
index domain : (forceVariableSetIndex,s)
range : free;
SET:
identifier : flowTAVariableSet
indices : flowTAVariableSetIndex
definition : data {'27' };
VARIABLE:
identifier : flowTAVariable
index domain : (flowTAVariableSetIndex,s)
range : free;
PARAMETER:
identifier : rpmMinParameter
definition : 1;
VARIABLE:
identifier : decisionVars
index domain : d
range : binary;
CONSTRAINT:
identifier : c0
index domain : s
definition : torque('16',s) = -tauVariable('35',s);
CONSTRAINT:
identifier : c1
index domain : s
definition : tauVariable('35',s) <= maxTauVariable('37',s);
CONSTRAINT:
identifier : co2
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('12') = 1
definition : omegaVariable('36',s) = (minSpeedParameter('1') +(maxSpeedParameter('1')-
minSpeedParameter('1'))*normalizedSpeedVariable('38',s));
SET:
identifier : lambdaSet1
indices : ls1
definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
PARAMETER:
identifier : lambdaWeights1
index domain : ls1
definition : data {1 : 1,2 : 2,3 : 3,4 : 4,5 : 5,6 : 6,7 : 7,8 : 8,9 : 9,10 : 10,11 : 11 };
VARIABLE:
identifier : lambdaVariable1
index domain : (ls1,s)
range : [0, 1];
PARAMETER:
identifier : normalizedSpeedInterpValues1
index domain : ls1
definition : data {1 : 0,2 : 0.1,3 : 0.2,4 : 0.3,5 : 0.4,6 : 0.5,7 : 0.6,8 : 0.7,9 : 0.8,10 : 0.9,11 : 1};
PARAMETER:
identifier : maxTauInterpValues1
```

```
596.  index domain : ls1
597.  definition : data {1 : 0, 2 : 2.34, 3 : 8.3, 4 : 13.4, 5 : 19, 6 : 35, 7 : 40, 8 : 42, 9 : 40, 10 : 35, 11 :
      20};
598.  CONSTRAINT:
599.  identifier : interpCom3
600.  index domain : s
601.  definition : maxTauVariable('37',s) = maxTauInterpValues1('1') * lambdaVariable1('1',s)
      +maxTauInterpValues1('2') * lambdaVariable1('2',s) +maxTauInterpValues1('3') *
      lambdaVariable1('3',s) +maxTauInterpValues1('4')
602.  lambdaVariable1('4',s) +maxTauInterpValues1('5') * lambdaVariable1('5',s)
      +maxTauInterpValues1('6') * lambdaVariable1('6',s) +maxTauInterpValues1('7') *
      lambdaVariable1('7',s) +maxTauInterpValues1('8')
603.  lambdaVariable1('8',s) +maxTauInterpValues1('9') * lambdaVariable1('9',s)
      +maxTauInterpValues1('10') * lambdaVariable1('10',s) +maxTauInterpValues1('11') *
      lambdaVariable1('11',s) ;
604.  CONSTRAINT:
605.  identifier : interpCom4
606.  index domain : s
607.  definition : normalizedSpeedVariable('38',s) = normalizedSpeedInterpValues1('1') *
      lambdaVariable1('1',s) +normalizedSpeedInterpValues1('2') * lambdaVariable1('2',s)
      +normalizedSpeedInterpValues1('3') * lambdaVariable1('3',s)
608.  +normalizedSpeedInterpValues1('4') * lambdaVariable1('4',s)
      +normalizedSpeedInterpValues1('5') * lambdaVariable1('5',s)
      +normalizedSpeedInterpValues1('6') * lambdaVariable1('6',s)
609.  +normalizedSpeedInterpValues1('7') * lambdaVariable1('7',s)
      +normalizedSpeedInterpValues1('8') * lambdaVariable1('8',s)
      +normalizedSpeedInterpValues1('9') * lambdaVariable1('9',s)
610.  +normalizedSpeedInterpValues1('10') * lambdaVariable1('10',s)
      +normalizedSpeedInterpValues1('11') * lambdaVariable1('11',s) ;
611.  CONSTRAINT:
612.  identifier : interpCom5
613.  index domain : s
614.  definition : lambdaVariable1('1',s) +lambdaVariable1('2',s) +lambdaVariable1('3',s)
      +lambdaVariable1('4',s) +lambdaVariable1('5',s) +lambdaVariable1('6',s)
      +lambdaVariable1('7',s) +lambdaVariable1('8',s) +lambdaVariable1('9',s)
615.  +lambdaVariable1('10',s) +lambdaVariable1('11',s) = cotsVar('12') ;
616.  CONSTRAINT:
617.  identifier : interpCom6
618.  index domain : s
619.  property : SOS2
620.  sos weight : lambdaVariable1(ls1,s) : lambdaWeights1(ls1)
621.  definition : lambdaVariable1('1',s) +lambdaVariable1('2',s) +lambdaVariable1('3',s)
      +lambdaVariable1('4',s) +lambdaVariable1('5',s) +lambdaVariable1('6',s)
      +lambdaVariable1('7',s) +lambdaVariable1('8',s) +lambdaVariable1('9',s)
622.  +lambdaVariable1('10',s) +lambdaVariable1('11',s) = 1 ;
623.  CONSTRAINT:
624.  identifier : c6
625.  index domain : s
626.  definition : angularVelocity('16',s) = -omegaVariable('36',s);
627.  PARAMETER:
628.  identifier : scalingXMin0
629.  definition : 0;
630.  PARAMETER:
631.  identifier : scalingXMax0
632.  definition : 50;
633.  PARAMETER:
```

```
634.  identifier : scalingYMin0
635.  definition : 107;
636.  PARAMETER:
637.  identifier : scalingYMax0
638.  definition : 300;
639.  SET:
640.  identifier : interpVarASet0
641.  indices : interpVarASetIndex0
642.  definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
643.  PARAMETER:
644.  identifier : interpVarAValues0
645.  index domain : interpVarASetIndex0
646.  definition : data {1 : -1.0,2 : -0.8,3 : -0.6000000000000001,4 : -0.4000000000000001,5 : -
      0.20000000000000007,6 : -5.551115123125783E-17,7 : 0.19999999999999996,8 :
      0.39999999999999997,9 : 0.6,10 : 0.8,11 : 1.0};
647.  SET:
648.  identifier : interpVarBSet0
649.  indices : interpVarBSetIndex0
650.  definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
651.  PARAMETER:
652.  identifier : interpVarBValues0
653.  index domain : interpVarBSetIndex0
654.  definition : data {1 : -1.0,2 : -0.8,3 : -0.6000000000000001,4 : -0.4000000000000001,5 : -
      0.20000000000000007,6 : -5.551115123125783E-17,7 : 0.19999999999999996,8 :
      0.39999999999999997,9 : 0.6,10 : 0.8,11 : 1.0};
655.  SET:
656.  identifier : interpVarA2Set0
657.  indices : interpVarA2SetIndex0
658.  definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
659.  PARAMETER:
660.  identifier : interpVarA2Values0
661.  index domain : interpVarA2SetIndex0
662.  definition : data {1 : 1.0,2 : 0.6400000000000001,3 : 0.3600000000000001,4 :
      0.16000000000000006,5 : 0.04000000000000003,6 : 3.0814879110195774E-33,7 :
      0.03999999999999998,8 : 0.15999999999999998,9 : 0.36,10 : 0.6400000000000001,11
663.  : 1.0} ;
664.  SET:
665.  identifier : interpVarB2Set0
666.  indices : interpVarB2SetIndex0
667.  definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
668.  PARAMETER:
669.  identifier : interpVarB2Values0
670.  index domain : interpVarB2SetIndex0
671.  definition : data {1 : -1.0,2 : -0.6400000000000001,3 : -0.3600000000000001,4 : -
      0.16000000000000006,5 : -0.04000000000000003,6 : -3.0814879110195774E-33,7 : -
      0.03999999999999998,8 : -0.15999999999999998,9 : -0.36,10
672.  : -0.6400000000000001,11 : -1.0} ;
673.  VARIABLE:
674.  identifier : interpVarA0
675.  index domain : s
676.  range : free;
677.  VARIABLE:
678.  identifier : interpVarB0
679.  index domain : s
680.  range : free;
681.  VARIABLE:
```

```
682.  identifier : scaledVarX0
683.  index domain : s
684.  range : free;
685.  VARIABLE:
686.  identifier : scaledVarY0
687.  index domain : s
688.  range : free;
689.  CONSTRAINT:
690.  identifier : scaledConA7
691.  index domain : s
692.  definition : tauVariable('35',s) = scaledVarX0(s) * (scalingXMax0- scalingXMin0) +
      scalingXMin0;
693.  CONSTRAINT:
694.  identifier : scaledConB7
695.  index domain : s
696.  definition : omegaVariable('36',s) = scaledVarY0(s) * (scalingYMax0- scalingYMin0) +
      scalingYMin0;
697.  CONSTRAINT:
698.  identifier : interpA7
699.  index domain : s
700.  definition : interpVarA0(s) = 1/2*(scaledVarX0(s) + scaledVarY0(s));
701.  CONSTRAINT:
702.  identifier : interpB7
703.  index domain : s
704.  definition : interpVarB0(s) = 1/2*(scaledVarX0(s) - scaledVarY0(s));
705.  VARIABLE:
706.  identifier : interpVarA20
707.  index domain : s
708.  range : free;
709.  VARIABLE:
710.  identifier : interpVarB20
711.  index domain : s
712.  range : free;
713.  SET:
714.  identifier : lambdaSet2
715.  indices : ls2
716.  definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
717.  VARIABLE:
718.  identifier : lambdaVariableA2
719.  index domain : (ls2,s)
720.  range : [0, 1];
721.  VARIABLE:
722.  identifier : lambdaVariableB2
723.  index domain : (ls2,s)
724.  range : [0, 1];
725.  CONSTRAINT:
726.  identifier : lambdaSumA8
727.  index domain : s
728.  definition : sum(ls2, lambdaVariableA2(ls2,s)) = 1;
729.  CONSTRAINT:
730.  identifier : lambdaSumB8
731.  index domain : s
732.  property : SOS2
733.  sos weight : lambdaVariableB2(ls2,s) :  interpVarBValues0(ls2)
734.  definition : sum(ls2, lambdaVariableB2(ls2,s)) = 1;
735.  CONSTRAINT:
```

identifier : interpVarACon8
index domain : s
definition : interpVarA0(s) = lambdaVariableA2('1',s) *
interpVarAValues0('1')+lambdaVariableA2('2',s) *
interpVarAValues0('2')+lambdaVariableA2('3',s) *
interpVarAValues0('3')+lambdaVariableA2('4',s) *
interpVarAValues0('4')+lambdaVariableA2('5',s)
interpVarAValues0('5')+lambdaVariableA2('6',s) *
interpVarAValues0('6')+lambdaVariableA2('7',s) *
interpVarAValues0('7')+lambdaVariableA2('8',s) *
interpVarAValues0('8')+lambdaVariableA2('9',s)
interpVarAValues0('9')+lambdaVariableA2('10',s) *
interpVarAValues0('10')+lambdaVariableA2('11',s) * interpVarAValues0('11') ;
CONSTRAINT:
identifier : interpVarBCon8
index domain : s
definition : interpVarB0(s) = lambdaVariableB2('1',s) *
interpVarBValues0('1')+lambdaVariableB2('2',s) *
interpVarBValues0('2')+lambdaVariableB2('3',s) *
interpVarBValues0('3')+lambdaVariableB2('4',s) *
interpVarBValues0('4')+lambdaVariableB2('5',s)
interpVarBValues0('5')+lambdaVariableB2('6',s) *
interpVarBValues0('6')+lambdaVariableB2('7',s) *
interpVarBValues0('7')+lambdaVariableB2('8',s) *
interpVarBValues0('8')+lambdaVariableB2('9',s)
interpVarBValues0('9')+lambdaVariableB2('10',s) *
interpVarBValues0('10')+lambdaVariableB2('11',s) * interpVarBValues0('11') ;
CONSTRAINT:
identifier : interpVarA2Con8
index domain : s
definition : interpVarA20(s) >= lambdaVariableA2('1',s) *
interpVarA2Values0('1')+lambdaVariableA2('2',s) *
interpVarA2Values0('2')+lambdaVariableA2('3',s) *
interpVarA2Values0('3')+lambdaVariableA2('4',s) *
interpVarA2Values0('4')+lambdaVariableA2('5',s)
interpVarA2Values0('5')+lambdaVariableA2('6',s) *
interpVarA2Values0('6')+lambdaVariableA2('7',s) *
interpVarA2Values0('7')+lambdaVariableA2('8',s)
interpVarA2Values0('8')+lambdaVariableA2('9',s) *
interpVarA2Values0('9')+lambdaVariableA2('10',s) *
interpVarA2Values0('10')+lambdaVariableA2('11',s) * interpVarA2Values0('11') ;
CONSTRAINT:
identifier : interpVarB2Con8
index domain : s
definition : interpVarB20(s) >= lambdaVariableB2('1',s) *
interpVarB2Values0('1')+lambdaVariableB2('2',s) *
interpVarB2Values0('2')+lambdaVariableB2('3',s) *
interpVarB2Values0('3')+lambdaVariableB2('4',s) *
interpVarB2Values0('4')+lambdaVariableB2('5',s)
interpVarB2Values0('5')+lambdaVariableB2('6',s) *
interpVarB2Values0('6')+lambdaVariableB2('7',s) *
interpVarB2Values0('7')+lambdaVariableB2('8',s)
interpVarB2Values0('8')+lambdaVariableB2('9',s) *
interpVarB2Values0('9')+lambdaVariableB2('10',s) *
interpVarB2Values0('10')+lambdaVariableB2('11',s) * interpVarB2Values0('11') ;
CONSTRAINT:

```
760.  identifier : c8
761.  index domain : s
762.  definition : powerVariable('34',s) >= (interpVarA20(s) + interpVarB20(s)) * (scalingXMax0 -
      scalingXMin0) * (scalingYMax0 - scalingYMin0) + scaledVarX0(s) * (scalingXMax0-
      scalingXMin0) * scalingYMin0 + scaledVarY0(s)
763.  (scalingYMax0 - scalingYMin0) * scalingXMin0 + scalingXMin0 * scalingYmin0 ;
764.  PARAMETER:
765.  identifier : scalingXMin1
766.  definition : 0;
767.  PARAMETER:
768.  identifier : scalingXMax1
769.  definition : .2;
770.  PARAMETER:
771.  identifier : scalingYMin1
772.  definition : 0;
773.  PARAMETER:
774.  identifier : scalingYMax1
775.  definition : 20000;
776.  SET:
777.  identifier : interpVarASet1
778.  indices : interpVarASetIndex1
779.  definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
780.  PARAMETER:
781.  identifier : interpVarAValues1
782.  index domain : interpVarASetIndex1
783.  definition : data {1 : -1.0,2 : -0.8,3 : -0.6000000000000001,4 : -0.4000000000000001,5 : -
      0.20000000000000007,6 : -5.551115123125783E-17,7 : 0.19999999999999996,8 :
      0.39999999999999997,9 : 0.6,10 : 0.8,11 : 1.0};
784.  SET:
785.  identifier : interpVarBSet1
786.  indices : interpVarBSetIndex1
787.  definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
788.  PARAMETER:
789.  identifier : interpVarBValues1
790.  index domain : interpVarBSetIndex1
791.  definition : data {1 : -1.0,2 : -0.8,3 : -0.6000000000000001,4 : -0.4000000000000001,5 : -
      0.20000000000000007,6 : -5.551115123125783E-17,7 : 0.19999999999999996,8 :
      0.39999999999999997,9 : 0.6,10 : 0.8,11 : 1.0};
792.  SET:
793.  identifier : interpVarA2Set1
794.  indices : interpVarA2SetIndex1
795.  definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
796.  PARAMETER:
797.  identifier : interpVarA2Values1
798.  index domain : interpVarA2SetIndex1
799.  definition : data {1 : 1.0,2 : 0.6400000000000001,3 : 0.3600000000000001,4 :
      0.16000000000000006,5 : 0.04000000000000003,6 : 3.0814879110195774E-33,7 :
      0.03999999999999998,8 : 0.15999999999999998,9 : 0.36,10 : 0.6400000000000001,11
800.  : 1.0} ;
801.  SET:
802.  identifier : interpVarB2Set1
803.  indices : interpVarB2SetIndex1
804.  definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
805.  PARAMETER:
806.  identifier : interpVarB2Values1
807.  index domain : interpVarB2SetIndex1
```

```
808.  definition : data {1 : -1.0,2 : -0.6400000000000001,3 : -0.3600000000000001,4 : -
      0.16000000000000006,5 : -0.04000000000000003,6 : -3.0814879110195774E-33,7 : -
      0.03999999999999998,8 : -0.15999999999999998,9 : -0.36,10
809.  : -0.6400000000000001,11 : -1.0} ;
810.  VARIABLE:
811.  identifier : interpVarA1
812.  index domain : s
813.  range : free;
814.  VARIABLE:
815.  identifier : interpVarB1
816.  index domain : s
817.  range : free;
818.  VARIABLE:
819.  identifier : scaledVarX1
820.  index domain : s
821.  range : free;
822.  VARIABLE:
823.  identifier : scaledVarY1
824.  index domain : s
825.  range : free;
826.  CONSTRAINT:
827.  identifier : scaledConA9
828.  index domain : s
829.  definition : fuelConsumptionVariable('39',s) = scaledVarX1(s) * (scalingXMax1-
      scalingXMin1) + scalingXMin1;
830.  CONSTRAINT:
831.  identifier : scaledConB9
832.  index domain : s
833.  definition : powerVariable('34',s) = scaledVarY1(s) * (scalingYMax1- scalingYMin1) +
      scalingYMin1;
834.  CONSTRAINT:
835.  identifier : interpA9
836.  index domain : s
837.  definition : interpVarA1(s) = 1/2*(scaledVarX1(s) + scaledVarY1(s));
838.  CONSTRAINT:
839.  identifier : interpB9
840.  index domain : s
841.  definition : interpVarB1(s) = 1/2*(scaledVarX1(s) - scaledVarY1(s));
842.  VARIABLE:
843.  identifier : interpVarA21
844.  index domain : s
845.  range : free;
846.  VARIABLE:
847.  identifier : interpVarB21
848.  index domain : s
849.  range : free;
850.  SET:
851.  identifier : lambdaSet4
852.  indices : ls4
853.  definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
854.  VARIABLE:
855.  identifier : lambdaVariableA4
856.  index domain : (ls4,s)
857.  range : [0, 1];
858.  VARIABLE:
859.  identifier : lambdaVariableB4
```

index domain : (ls4,s)
range : [0, 1];
CONSTRAINT:
identifier : lambdaSumA10
index domain : s
definition : sum(ls4, lambdaVariableA4(ls4,s)) = 1;
CONSTRAINT:
identifier : lambdaSumB10
index domain : s
property : SOS2
sos weight : lambdaVariableB4(ls4,s) : interpVarBValues1(ls4)
definition : sum(ls4, lambdaVariableB4(ls4,s)) = 1;
CONSTRAINT:
identifier : interpVarACon10
index domain : s
definition : interpVarA1(s) = lambdaVariableA4('1',s) *
interpVarAValues1('1')+lambdaVariableA4('2',s) *
interpVarAValues1('2')+lambdaVariableA4('3',s) *
interpVarAValues1('3')+lambdaVariableA4('4',s) *
interpVarAValues1('4')+lambdaVariableA4('5',s)
interpVarAValues1('5')+lambdaVariableA4('6',s) *
interpVarAValues1('6')+lambdaVariableA4('7',s) *
interpVarAValues1('7')+lambdaVariableA4('8',s) *
interpVarAValues1('8')+lambdaVariableA4('9',s)
interpVarAValues1('9')+lambdaVariableA4('10',s) *
interpVarAValues1('10')+lambdaVariableA4('11',s) * interpVarAValues1('11') ;
CONSTRAINT:
identifier : interpVarBCon10
index domain : s
definition : interpVarB1(s) = lambdaVariableB4('1',s) *
interpVarBValues1('1')+lambdaVariableB4('2',s) *
interpVarBValues1('2')+lambdaVariableB4('3',s) *
interpVarBValues1('3')+lambdaVariableB4('4',s) *
interpVarBValues1('4')+lambdaVariableB4('5',s)
interpVarBValues1('5')+lambdaVariableB4('6',s) *
interpVarBValues1('6')+lambdaVariableB4('7',s) *
interpVarBValues1('7')+lambdaVariableB4('8',s) *
interpVarBValues1('8')+lambdaVariableB4('9',s)
interpVarBValues1('9')+lambdaVariableB4('10',s) *
interpVarBValues1('10')+lambdaVariableB4('11',s) * interpVarBValues1('11') ;
CONSTRAINT:
identifier : interpVarA2Con10
index domain : s
definition : interpVarA21(s) >= lambdaVariableA4('1',s) *
interpVarA2Values1('1')+lambdaVariableA4('2',s) *
interpVarA2Values1('2')+lambdaVariableA4('3',s) *
interpVarA2Values1('3')+lambdaVariableA4('4',s) *
interpVarA2Values1('4')+lambdaVariableA4('5',s)
interpVarA2Values1('5')+lambdaVariableA4('6',s) *
interpVarA2Values1('6')+lambdaVariableA4('7',s) *
interpVarA2Values1('7')+lambdaVariableA4('8',s)
interpVarA2Values1('8')+lambdaVariableA4('9',s) *
interpVarA2Values1('9')+lambdaVariableA4('10',s) *
interpVarA2Values1('10')+lambdaVariableA4('11',s) * interpVarA2Values1('11') ;
CONSTRAINT:
identifier : interpVarB2Con10

```
892.  index domain : s
893.  definition : interpVarB21(s) >= lambdaVariableB4('1',s) *
      interpVarB2Values1('1')+lambdaVariableB4('2',s) *
      interpVarB2Values1('2')+lambdaVariableB4('3',s) *
      interpVarB2Values1('3')+lambdaVariableB4('4',s) *
      interpVarB2Values1('4')+lambdaVariableB4('5',s)
894.  interpVarB2Values1('5')+lambdaVariableB4('6',s) *
      interpVarB2Values1('6')+lambdaVariableB4('7',s) *
      interpVarB2Values1('7')+lambdaVariableB4('8',s)
895.  interpVarB2Values1('8')+lambdaVariableB4('9',s) *
      interpVarB2Values1('9')+lambdaVariableB4('10',s) *
      interpVarB2Values1('10')+lambdaVariableB4('11',s) * interpVarB2Values1('11') ;
896.  CONSTRAINT:
897.  identifier : c10
898.  index domain : s
899.  definition : fuelRateVariable('40',s) >= (interpVarA21(s) + interpVarB21(s)) * (scalingXMax1
      - scalingXMin1) * (scalingYMax1 - scalingYMin1) + scaledVarX1(s) * (scalingXMax1-
      scalingXMin1) * scalingYMin1 + scaledVarY1(s)
900.  (scalingYMax1 - scalingYMin1) * scalingXMin1 + scalingXMin1 * scalingYmin1 ;
901.  SET:
902.  identifier : lambdaSet6
903.  indices : ls6
904.  definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
905.  PARAMETER:
906.  identifier : lambdaWeights6
907.  index domain : ls6
908.  definition : data {1 : 1,2 : 2,3 : 3,4 : 4,5 : 5,6 : 6,7 : 7,8 : 8,9 : 9,10 : 10,11 : 11 };
909.  VARIABLE:
910.  identifier : lambdaVariable6
911.  index domain : (ls6,s)
912.  range : [0, 1];
913.  PARAMETER:
914.  identifier : normalizedSpeedInterpValues6
915.  index domain : ls6
916.  definition : data {1 : 0,2 : 0.1,3 : 0.2,4 : 0.3,5 : 0.4,6 : 0.5,7 : 0.6,8 : 0.7,9 : 0.8,10 : 0.9,11 : 1};
917.  PARAMETER:
918.  identifier : fuelConsumptionInterpValues6
919.  index domain : ls6
920.  definition : data {1 : 0.219, 2 : 0.21, 3 : 0.2, 4 : 0.2, 5 : 0.2, 6 : 0.194, 7 : 0.2, 8 : 0.2, 9 : 0.2, 10
      : 0.21, 11 : 0.219};
921.  CONSTRAINT:
922.  identifier : interpCom11
923.  index domain : s
924.  definition : fuelConsumptionVariable('39',s) = fuelConsumptionInterpValues6('1') *
      lambdaVariable6('1',s) +fuelConsumptionInterpValues6('2') * lambdaVariable6('2',s)
      +fuelConsumptionInterpValues6('3') * lambdaVariable6('3',s)
925.  +fuelConsumptionInterpValues6('4') * lambdaVariable6('4',s)
      +fuelConsumptionInterpValues6('5') * lambdaVariable6('5',s)
      +fuelConsumptionInterpValues6('6') * lambdaVariable6('6',s)
926.  +fuelConsumptionInterpValues6('7') * lambdaVariable6('7',s)
      +fuelConsumptionInterpValues6('8') * lambdaVariable6('8',s)
      +fuelConsumptionInterpValues6('9') * lambdaVariable6('9',s)
927.  +fuelConsumptionInterpValues6('10') * lambdaVariable6('10',s)
      +fuelConsumptionInterpValues6('11') * lambdaVariable6('11',s) ;
928.  CONSTRAINT:
929.  identifier : interpCom12
```

```
930.  index domain : s
931.  definition : normalizedSpeedVariable('38',s) = normalizedSpeedInterpValues6('1') *
      lambdaVariable6('1',s) +normalizedSpeedInterpValues6('2') * lambdaVariable6('2',s)
      +normalizedSpeedInterpValues6('3') * lambdaVariable6('3',s)
932.  +normalizedSpeedInterpValues6('4') * lambdaVariable6('4',s)
      +normalizedSpeedInterpValues6('5') * lambdaVariable6('5',s)
      +normalizedSpeedInterpValues6('6') * lambdaVariable6('6',s)
933.  +normalizedSpeedInterpValues6('7') * lambdaVariable6('7',s)
      +normalizedSpeedInterpValues6('8') * lambdaVariable6('8',s)
      +normalizedSpeedInterpValues6('9') * lambdaVariable6('9',s)
934.  +normalizedSpeedInterpValues6('10') * lambdaVariable6('10',s)
      +normalizedSpeedInterpValues6('11') * lambdaVariable6('11',s) ;
935.  CONSTRAINT:
936.  identifier : interpCom13
937.  index domain : s
938.  definition : lambdaVariable6('1',s) +lambdaVariable6('2',s) +lambdaVariable6('3',s)
      +lambdaVariable6('4',s) +lambdaVariable6('5',s) +lambdaVariable6('6',s)
      +lambdaVariable6('7',s) +lambdaVariable6('8',s) +lambdaVariable6('9',s)
939.  +lambdaVariable6('10',s) +lambdaVariable6('11',s) = cotsVar('12') ;
940.  CONSTRAINT:
941.  identifier : interpCom14
942.  index domain : s
943.  property : SOS2
944.  sos weight : lambdaVariable6(ls6,s) : lambdaWeights6(ls6)
945.  definition : lambdaVariable6('1',s) +lambdaVariable6('2',s) +lambdaVariable6('3',s)
      +lambdaVariable6('4',s) +lambdaVariable6('5',s) +lambdaVariable6('6',s)
      +lambdaVariable6('7',s) +lambdaVariable6('8',s) +lambdaVariable6('9',s)
946.  +lambdaVariable6('10',s) +lambdaVariable6('11',s) = 1 ;
947.  CONSTRAINT:
948.  identifier : c14
949.  index domain : s
950.  definition : tauVariable('35',s) >= 0;
951.  CONSTRAINT:
952.  identifier : c15
953.  index domain : s
954.  definition : flow('2',s) + flow('3',s) = 0;
955.  PARAMETER:
956.  identifier : scalingXMin2
957.  definition : 0;
958.  PARAMETER:
959.  identifier : scalingXMax2
960.  definition : 1e8;
961.  PARAMETER:
962.  identifier : scalingYMin2
963.  definition : -250;
964.  PARAMETER:
965.  identifier : scalingYMax2
966.  definition : 250;
967.  SET:
968.  identifier : interpVarASet2
969.  indices : interpVarASetIndex2
970.  definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
971.  PARAMETER:
972.  identifier : interpVarAValues2
973.  index domain : interpVarASetIndex2
```

```
974.  definition : data {1 : -1.0,2 : -0.8,3 : -0.6000000000000001,4 : -0.4000000000000001,5 : -
      0.20000000000000007,6 : -5.551115123125783E-17,7 : 0.19999999999999996,8 :
      0.39999999999999997,9 : 0.6,10 : 0.8,11 : 1.0};
975.  SET:
976.  identifier : interpVarBSet2
977.  indices : interpVarBSetIndex2
978.  definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
979.  PARAMETER:
980.  identifier : interpVarBValues2
981.  index domain : interpVarBSetIndex2
982.  definition : data {1 : -1.0,2 : -0.8,3 : -0.6000000000000001,4 : -0.4000000000000001,5 : -
      0.20000000000000007,6 : -5.551115123125783E-17,7 : 0.19999999999999996,8 :
      0.39999999999999997,9 : 0.6,10 : 0.8,11 : 1.0};
983.  SET:
984.  identifier : interpVarA2Set2
985.  indices : interpVarA2SetIndex2
986.  definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
987.  PARAMETER:
988.  identifier : interpVarA2Values2
989.  index domain : interpVarA2SetIndex2
990.  definition : data {1 : 1.0,2 : 0.6400000000000001,3 : 0.3600000000000001,4 :
      0.16000000000000006,5 : 0.04000000000000003,6 : 3.0814879110195774E-33,7 :
      0.03999999999999998,8 : 0.15999999999999998,9 : 0.36,10 : 0.6400000000000001,11
991.  : 1.0} ;
992.  SET:
993.  identifier : interpVarB2Set2
994.  indices : interpVarB2SetIndex2
995.  definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
996.  PARAMETER:
997.  identifier : interpVarB2Values2
998.  index domain : interpVarB2SetIndex2
999.  definition : data {1 : 1.0,2 : 0.6400000000000001,3 : 0.3600000000000001,4 :
      0.16000000000000006,5 : 0.04000000000000003,6 : 3.0814879110195774E-33,7 :
      0.03999999999999998,8 : 0.15999999999999998,9 : 0.36,10 : 0.6400000000000001,11
1000. : 1.0} ;
1001. VARIABLE:
1002. identifier : interpVarA2
1003. index domain : s
1004. range : free;
1005. VARIABLE:
1006. identifier : interpVarB2
1007. index domain : s
1008. range : free;
1009. VARIABLE:
1010. identifier : scaledVarX2
1011. index domain : s
1012. range : free;
1013. VARIABLE:
1014. identifier : scaledVarY2
1015. index domain : s
1016. range : free;
1017. CONSTRAINT:
1018. identifier : scaledConA16
1019. index domain : s
1020. definition : prdiffVariable('13',s) = scaledVarX2(s) * (scalingXMax2- scalingXMin2) +
      scalingXMin2;
```

```
CONSTRAINT:
identifier : scaledConB16
index domain : s
definition : omegaVariable('12',s) = scaledVarY2(s) * (scalingYMax2- scalingYMin2) +
scalingYMin2;
CONSTRAINT:
identifier : interpA16
index domain : s
definition : interpVarA2(s) = 1/2*(scaledVarX2(s) + scaledVarY2(s));
CONSTRAINT:
identifier : interpB16
index domain : s
definition : interpVarB2(s) = 1/2*(scaledVarX2(s) - scaledVarY2(s));
VARIABLE:
identifier : interpVarA22
index domain : s
range : free;
VARIABLE:
identifier : interpVarB22
index domain : s
range : free;
SET:
identifier : lambdaSet7
indices : ls7
definition : data {'1','2','3','4','5','6','7','8','9','10','11' };
VARIABLE:
identifier : lambdaVariableA7
index domain : (ls7,s)
range : [0, 1];
VARIABLE:
identifier : lambdaVariableB7
index domain : (ls7,s)
range : [0, 1];
CONSTRAINT:
identifier : lambdaSumA17
index domain : s
property : SOS2
sos weight : lambdaVariableA7(ls7,s) :  interpVarAValues2(ls7)
definition : sum(ls7, lambdaVariableA7(ls7,s)) = 1;
CONSTRAINT:
identifier : lambdaSumB17
index domain : s
property : SOS2
sos weight : lambdaVariableB7(ls7,s) :  interpVarBValues2(ls7)
definition : sum(ls7, lambdaVariableB7(ls7,s)) = 1;
CONSTRAINT:
identifier : interpVarACon17
index domain : s
definition : interpVarA2(s) = lambdaVariableA7('1',s) *
interpVarAValues2('1')+lambdaVariableA7('2',s) *
interpVarAValues2('2')+lambdaVariableA7('3',s) *
interpVarAValues2('3')+lambdaVariableA7('4',s) *
interpVarAValues2('4')+lambdaVariableA7('5',s)
interpVarAValues2('5')+lambdaVariableA7('6',s) *
interpVarAValues2('6')+lambdaVariableA7('7',s) *
```

interpVarAValues2('7')+lambdaVariableA7('8',s) *
interpVarAValues2('8')+lambdaVariableA7('9',s)
1070. interpVarAValues2('9')+lambdaVariableA7('10',s) *
interpVarAValues2('10')+lambdaVariableA7('11',s) * interpVarAValues2('11') ;
1071. CONSTRAINT:
1072. identifier : interpVarBCon17
1073. index domain : s
1074. definition : interpVarB2(s) = lambdaVariableB7('1',s) *
interpVarBValues2('1')+lambdaVariableB7('2',s) *
interpVarBValues2('2')+lambdaVariableB7('3',s) *
interpVarBValues2('3')+lambdaVariableB7('4',s) *
interpVarBValues2('4')+lambdaVariableB7('5',s)
1075. interpVarBValues2('5')+lambdaVariableB7('6',s) *
interpVarBValues2('6')+lambdaVariableB7('7',s) *
interpVarBValues2('7')+lambdaVariableB7('8',s) *
interpVarBValues2('8')+lambdaVariableB7('9',s)
1076. interpVarBValues2('9')+lambdaVariableB7('10',s) *
interpVarBValues2('10')+lambdaVariableB7('11',s) * interpVarBValues2('11') ;
1077. CONSTRAINT:
1078. identifier : interpVarA2Con17
1079. index domain : s
1080. definition : interpVarA22(s) = lambdaVariableA7('1',s) *
interpVarA2Values2('1')+lambdaVariableA7('2',s) *
interpVarA2Values2('2')+lambdaVariableA7('3',s) *
interpVarA2Values2('3')+lambdaVariableA7('4',s) *
interpVarA2Values2('4')+lambdaVariableA7('5',s)
1081. interpVarA2Values2('5')+lambdaVariableA7('6',s) *
interpVarA2Values2('6')+lambdaVariableA7('7',s) *
interpVarA2Values2('7')+lambdaVariableA7('8',s)
1082. interpVarA2Values2('8')+lambdaVariableA7('9',s) *
interpVarA2Values2('9')+lambdaVariableA7('10',s) *
interpVarA2Values2('10')+lambdaVariableA7('11',s) * interpVarA2Values2('11') ;
1083. CONSTRAINT:
1084. identifier : interpVarB2Con17
1085. index domain : s
1086. definition : interpVarB22(s) = lambdaVariableB7('1',s) *
interpVarB2Values2('1')+lambdaVariableB7('2',s) *
interpVarB2Values2('2')+lambdaVariableB7('3',s) *
interpVarB2Values2('3')+lambdaVariableB7('4',s) *
interpVarB2Values2('4')+lambdaVariableB7('5',s)
1087. interpVarB2Values2('5')+lambdaVariableB7('6',s) *
interpVarB2Values2('6')+lambdaVariableB7('7',s) *
interpVarB2Values2('7')+lambdaVariableB7('8',s)
1088. interpVarB2Values2('8')+lambdaVariableB7('9',s) *
interpVarB2Values2('9')+lambdaVariableB7('10',s) *
interpVarB2Values2('10')+lambdaVariableB7('11',s) * interpVarB2Values2('11') ;
1089. CONSTRAINT:
1090. identifier : c17
1091. index domain : s
1092. definition : pOmegaVariable('8',s) = (interpVarA22(s) - interpVarB22(s)) * (scalingXMax2 -
scalingXMin2) * (scalingYMax2 - scalingYMin2) + scaledVarX2(s) * (scalingXMax2-
scalingXMin2) * scalingYMin2 + scaledVarY2(s)
1093. (scalingYMax2 - scalingYMin2) * scalingXMin2 + scalingXMin2 * scalingYmin2 ;
1094. CONSTRAINT:
1095. identifier : co18
1096. index domain : s

property : IndicatorConstraint
activating condition : cotsVar('1') = 1
definition : qLossPVariable('6',s) = csParameter('1') * prdiffVariable('13',s) * (1/muParameter('1'));
CONSTRAINT:
identifier : co19
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('2') = 1
definition : qLossPVariable('6',s) = csParameter('2') * prdiffVariable('13',s) * (1/muParameter('2'));
CONSTRAINT:
identifier : co20
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('3') = 1
definition : qLossPVariable('6',s) = csParameter('3') * prdiffVariable('13',s) * (1/muParameter('3'));
CONSTRAINT:
identifier : co21
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('4') = 1
definition : qLossPVariable('6',s) = csParameter('4') * prdiffVariable('13',s) * (1/muParameter('4'));
CONSTRAINT:
identifier : co22
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('5') = 1
definition : qLossPVariable('6',s) = csParameter('5') * prdiffVariable('13',s) * (1/muParameter('5'));
CONSTRAINT:
identifier : co23
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('6') = 1
definition : qLossPVariable('6',s) = csParameter('6') * prdiffVariable('13',s) * (1/muParameter('6'));
CONSTRAINT:
identifier : co24
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('7') = 1
definition : qLossPVariable('6',s) = csParameter('7') * prdiffVariable('13',s) * (1/muParameter('7'));
CONSTRAINT:
identifier : co25
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('8') = 1
definition : qLossPVariable('6',s) = csParameter('8') * prdiffVariable('13',s) * (1/muParameter('8'));
CONSTRAINT:
identifier : co26
index domain : s

property : IndicatorConstraint
activating condition : cotsVar('9') = 1
definition : qLossPVariable('6',s) = csParameter('9') * prdiffVariable('13',s) * (1/muParameter('9'));
CONSTRAINT:
identifier : co27
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('10') = 1
definition : qLossPVariable('6',s) = csParameter('10') * prdiffVariable('13',s) * (1/muParameter('10'));
CONSTRAINT:
identifier : co28
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('11') = 1
definition : qLossPVariable('6',s) = csParameter('11') * prdiffVariable('13',s) * (1/muParameter('11'));
CONSTRAINT:
identifier : co29
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('1') = 1
definition : tLossOmegaVariable('14',s) >= cvParameter('1') * muParameter('1') * omegaVariable('12',s);
CONSTRAINT:
identifier : co30
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('2') = 1
definition : tLossOmegaVariable('14',s) >= cvParameter('2') * muParameter('2') * omegaVariable('12',s);
CONSTRAINT:
identifier : co31
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('3') = 1
definition : tLossOmegaVariable('14',s) >= cvParameter('3') * muParameter('3') * omegaVariable('12',s);
CONSTRAINT:
identifier : co32
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('4') = 1
definition : tLossOmegaVariable('14',s) >= cvParameter('4') * muParameter('4') * omegaVariable('12',s);
CONSTRAINT:
identifier : co33
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('5') = 1
definition : tLossOmegaVariable('14',s) >= cvParameter('5') * muParameter('5') * omegaVariable('12',s);
CONSTRAINT:
identifier : co34
index domain : s

property : IndicatorConstraint
activating condition : cotsVar('6') = 1
definition : tLossOmegaVariable('14',s) >= cvParameter('6') * muParameter('6') * omegaVariable('12',s);
CONSTRAINT:
identifier : co35
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('7') = 1
definition : tLossOmegaVariable('14',s) >= cvParameter('7') * muParameter('7') * omegaVariable('12',s);
CONSTRAINT:
identifier : co36
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('8') = 1
definition : tLossOmegaVariable('14',s) >= cvParameter('8') * muParameter('8') * omegaVariable('12',s);
CONSTRAINT:
identifier : co37
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('9') = 1
definition : tLossOmegaVariable('14',s) >= cvParameter('9') * muParameter('9') * omegaVariable('12',s);
CONSTRAINT:
identifier : co38
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('10') = 1
definition : tLossOmegaVariable('14',s) >= cvParameter('10') * muParameter('10') * omegaVariable('12',s);
CONSTRAINT:
identifier : co39
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('11') = 1
definition : tLossOmegaVariable('14',s) >= cvParameter('11') * muParameter('11') * omegaVariable('12',s);
CONSTRAINT:
identifier : co40
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('1') = 1
definition : tLossPVariable('5',s) >= cfParameter('1') * prdiffVariable('13',s);
CONSTRAINT:
identifier : co41
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('2') = 1
definition : tLossPVariable('5',s) >= cfParameter('2') * prdiffVariable('13',s);
CONSTRAINT:
identifier : co42
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('3') = 1

definition : tLossPVariable('5',s) >= cfParameter('3') * prdiffVariable('13',s);
CONSTRAINT:
identifier : co43
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('4') = 1
definition : tLossPVariable('5',s) >= cfParameter('4') * prdiffVariable('13',s);
CONSTRAINT:
identifier : co44
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('5') = 1
definition : tLossPVariable('5',s) >= cfParameter('5') * prdiffVariable('13',s);
CONSTRAINT:
identifier : co45
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('6') = 1
definition : tLossPVariable('5',s) >= cfParameter('6') * prdiffVariable('13',s);
CONSTRAINT:
identifier : co46
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('7') = 1
definition : tLossPVariable('5',s) >= cfParameter('7') * prdiffVariable('13',s);
CONSTRAINT:
identifier : co47
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('8') = 1
definition : tLossPVariable('5',s) >= cfParameter('8') * prdiffVariable('13',s);
CONSTRAINT:
identifier : co48
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('9') = 1
definition : tLossPVariable('5',s) >= cfParameter('9') * prdiffVariable('13',s);
CONSTRAINT:
identifier : co49
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('10') = 1
definition : tLossPVariable('5',s) >= cfParameter('10') * prdiffVariable('13',s);
CONSTRAINT:
identifier : co50
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('11') = 1
definition : tLossPVariable('5',s) >= cfParameter('11') * prdiffVariable('13',s);
CONSTRAINT:
identifier : c51
index domain : s
definition : pressure('2',s)=prdiffVariable('13',s);
CONSTRAINT:
identifier : co52
index domain : s

property : IndicatorConstraint
activating condition : cotsVar('1') = 1
definition : qLossOmegaPVariable('2',s)=-
(pOmegaVariable('8',s)/BParameter('1'))*(VrParameter+1);
CONSTRAINT:
identifier : co53
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('2') = 1
definition : qLossOmegaPVariable('2',s)=-
(pOmegaVariable('8',s)/BParameter('2'))*(VrParameter+1);
CONSTRAINT:
identifier : co54
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('3') = 1
definition : qLossOmegaPVariable('2',s)=-
(pOmegaVariable('8',s)/BParameter('3'))*(VrParameter+1);
CONSTRAINT:
identifier : co55
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('4') = 1
definition : qLossOmegaPVariable('2',s)=-
(pOmegaVariable('8',s)/BParameter('4'))*(VrParameter+1);
CONSTRAINT:
identifier : co56
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('5') = 1
definition : qLossOmegaPVariable('2',s)=-
(pOmegaVariable('8',s)/BParameter('5'))*(VrParameter+1);
CONSTRAINT:
identifier : co57
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('6') = 1
definition : qLossOmegaPVariable('2',s)=-
(pOmegaVariable('8',s)/BParameter('6'))*(VrParameter+1);
CONSTRAINT:
identifier : co58
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('7') = 1
definition : qLossOmegaPVariable('2',s)=-
(pOmegaVariable('8',s)/BParameter('7'))*(VrParameter+1);
CONSTRAINT:
identifier : co59
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('8') = 1
definition : qLossOmegaPVariable('2',s)=-
(pOmegaVariable('8',s)/BParameter('8'))*(VrParameter+1);
CONSTRAINT:
identifier : co60
index domain : s

property : IndicatorConstraint
activating condition : cotsVar('9') = 1
definition : qLossOmegaPVariable('2',s)=-(pOmegaVariable('8',s)/BParameter('9'))*(VrParameter+1);
CONSTRAINT:
identifier : co61
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('10') = 1
definition : qLossOmegaPVariable('2',s)=-(pOmegaVariable('8',s)/BParameter('10'))*(VrParameter+1);
CONSTRAINT:
identifier : co62
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('11') = 1
definition : qLossOmegaPVariable('2',s)=-(pOmegaVariable('8',s)/BParameter('11'))*(VrParameter+1);
CONSTRAINT:
identifier : c63
index domain : s
definition : angularVelocity('1',s)=-omegaVariable('12',s);
CONSTRAINT:
identifier : co64
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('1') = 1
definition : torque('1',s) = dispParameter('1') * (prdiffVariable('13',s) + tLossPVariable('5',s)+tLossOmegaVariable('14',s));
CONSTRAINT:
identifier : co65
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('2') = 1
definition : torque('1',s) = dispParameter('2') * (prdiffVariable('13',s) + tLossPVariable('5',s)+tLossOmegaVariable('14',s));
CONSTRAINT:
identifier : co66
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('3') = 1
definition : torque('1',s) = dispParameter('3') * (prdiffVariable('13',s) + tLossPVariable('5',s)+tLossOmegaVariable('14',s));
CONSTRAINT:
identifier : co67
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('4') = 1
definition : torque('1',s) = dispParameter('4') * (prdiffVariable('13',s) + tLossPVariable('5',s)+tLossOmegaVariable('14',s));
CONSTRAINT:
identifier : co68
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('5') = 1

definition : torque('1',s) = dispParameter('5') * (prdiffVariable('13',s) + tLossPVariable('5',s)+tLossOmegaVariable('14',s));
CONSTRAINT:
identifier : co69
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('6') = 1
definition : torque('1',s) = dispParameter('6') * (prdiffVariable('13',s) + tLossPVariable('5',s)+tLossOmegaVariable('14',s));
CONSTRAINT:
identifier : co70
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('7') = 1
definition : torque('1',s) = dispParameter('7') * (prdiffVariable('13',s) + tLossPVariable('5',s)+tLossOmegaVariable('14',s));
CONSTRAINT:
identifier : co71
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('8') = 1
definition : torque('1',s) = dispParameter('8') * (prdiffVariable('13',s) + tLossPVariable('5',s)+tLossOmegaVariable('14',s));
CONSTRAINT:
identifier : co72
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('9') = 1
definition : torque('1',s) = dispParameter('9') * (prdiffVariable('13',s) + tLossPVariable('5',s)+tLossOmegaVariable('14',s));
CONSTRAINT:
identifier : co73
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('10') = 1
definition : torque('1',s) = dispParameter('10') * (prdiffVariable('13',s) + tLossPVariable('5',s)+tLossOmegaVariable('14',s));
CONSTRAINT:
identifier : co74
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('11') = 1
definition : torque('1',s) = dispParameter('11') * (prdiffVariable('13',s) + tLossPVariable('5',s)+tLossOmegaVariable('14',s));
CONSTRAINT:
identifier : co75
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('1') = 1
definition : dispParameter('1')*(angularVelocity('1',s)*2*pi- qLossPVariable('6',s) - qLossOmegaPVariable('2',s)) = flow('2',s);
CONSTRAINT:
identifier : co76
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('2') = 1

1443. definition : dispParameter('2')*(angularVelocity('1',s)*2*pi- qLossPVariable('6',s) - qLossOmegaPVariable('2',s)) = flow('2',s);
1444. CONSTRAINT:
1445. identifier : co77
1446. index domain : s
1447. property : IndicatorConstraint
1448. activating condition : cotsVar('3') = 1
1449. definition : dispParameter('3')*(angularVelocity('1',s)*2*pi- qLossPVariable('6',s) - qLossOmegaPVariable('2',s)) = flow('2',s);
1450. CONSTRAINT:
1451. identifier : co78
1452. index domain : s
1453. property : IndicatorConstraint
1454. activating condition : cotsVar('4') = 1
1455. definition : dispParameter('4')*(angularVelocity('1',s)*2*pi- qLossPVariable('6',s) - qLossOmegaPVariable('2',s)) = flow('2',s);
1456. CONSTRAINT:
1457. identifier : co79
1458. index domain : s
1459. property : IndicatorConstraint
1460. activating condition : cotsVar('5') = 1
1461. definition : dispParameter('5')*(angularVelocity('1',s)*2*pi- qLossPVariable('6',s) - qLossOmegaPVariable('2',s)) = flow('2',s);
1462. CONSTRAINT:
1463. identifier : co80
1464. index domain : s
1465. property : IndicatorConstraint
1466. activating condition : cotsVar('6') = 1
1467. definition : dispParameter('6')*(angularVelocity('1',s)*2*pi- qLossPVariable('6',s) - qLossOmegaPVariable('2',s)) = flow('2',s);
1468. CONSTRAINT:
1469. identifier : co81
1470. index domain : s
1471. property : IndicatorConstraint
1472. activating condition : cotsVar('7') = 1
1473. definition : dispParameter('7')*(angularVelocity('1',s)*2*pi- qLossPVariable('6',s) - qLossOmegaPVariable('2',s)) = flow('2',s);
1474. CONSTRAINT:
1475. identifier : co82
1476. index domain : s
1477. property : IndicatorConstraint
1478. activating condition : cotsVar('8') = 1
1479. definition : dispParameter('8')*(angularVelocity('1',s)*2*pi- qLossPVariable('6',s) - qLossOmegaPVariable('2',s)) = flow('2',s);
1480. CONSTRAINT:
1481. identifier : co83
1482. index domain : s
1483. property : IndicatorConstraint
1484. activating condition : cotsVar('9') = 1
1485. definition : dispParameter('9')*(angularVelocity('1',s)*2*pi- qLossPVariable('6',s) - qLossOmegaPVariable('2',s)) = flow('2',s);
1486. CONSTRAINT:
1487. identifier : co84
1488. index domain : s
1489. property : IndicatorConstraint
1490. activating condition : cotsVar('10') = 1

definition : dispParameter('10')*(angularVelocity('1',s)*2*pi- qLossPVariable('6',s) - qLossOmegaPVariable('2',s)) = flow('2',s);
CONSTRAINT:
identifier : co85
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('11') = 1
definition : dispParameter('11')*(angularVelocity('1',s)*2*pi- qLossPVariable('6',s) - qLossOmegaPVariable('2',s)) = flow('2',s);
CONSTRAINT:
identifier : c86
index domain : s
definition : pressure('21',s)=0;
CONSTRAINT:
identifier : c87
index domain : s
definition : pressure('22',s)=0;
CONSTRAINT:
identifier : c88
index domain : s
definition : flow('6',s)=0;
CONSTRAINT:
identifier : c89
index domain : s
definition : flow('7',s)=0;
CONSTRAINT:
identifier : c90
index domain : s
definition : pressure('4',s)=0;
CONSTRAINT:
identifier : c91
index domain : s
definition : pressure('5',s)=0;
CONSTRAINT:
identifier : co92
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('13') = 1
definition : pressure('17',s)*boreAreaParameter('1')-pressure('18',s)*rodAreaParameter('1') + force('19',s) = 0;
CONSTRAINT:
identifier : co93
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('14') = 1
definition : pressure('17',s)*boreAreaParameter('2')-pressure('18',s)*rodAreaParameter('2') + force('19',s) = 0;
CONSTRAINT:
identifier : co94
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('15') = 1
definition : pressure('17',s)*boreAreaParameter('3')-pressure('18',s)*rodAreaParameter('3') + force('19',s) = 0;
CONSTRAINT:
identifier : co95

index domain : s
property : IndicatorConstraint
activating condition : cotsVar('16') = 1
definition : pressure('17',s)*boreAreaParameter('4')-pressure('18',s)*rodAreaParameter('4') + force('19',s) = 0;
CONSTRAINT:
identifier : co96
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('17') = 1
definition : pressure('17',s)*boreAreaParameter('5')-pressure('18',s)*rodAreaParameter('5') + force('19',s) = 0;
CONSTRAINT:
identifier : co97
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('18') = 1
definition : pressure('17',s)*boreAreaParameter('6')-pressure('18',s)*rodAreaParameter('6') + force('19',s) = 0;
CONSTRAINT:
identifier : co98
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('19') = 1
definition : pressure('17',s)*boreAreaParameter('7')-pressure('18',s)*rodAreaParameter('7') + force('19',s) = 0;
CONSTRAINT:
identifier : co99
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('13') = 1
definition : flow('17',s)*boreAreaParameter('1')+flow('18',s)*rodAreaParameter('1') = 0;
CONSTRAINT:
identifier : co100
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('14') = 1
definition : flow('17',s)*boreAreaParameter('2')+flow('18',s)*rodAreaParameter('2') = 0;
CONSTRAINT:
identifier : co101
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('15') = 1
definition : flow('17',s)*boreAreaParameter('3')+flow('18',s)*rodAreaParameter('3') = 0;
CONSTRAINT:
identifier : co102
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('16') = 1
definition : flow('17',s)*boreAreaParameter('4')+flow('18',s)*rodAreaParameter('4') = 0;
CONSTRAINT:
identifier : co103
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('17') = 1
definition : flow('17',s)*boreAreaParameter('5')+flow('18',s)*rodAreaParameter('5') = 0;

```
CONSTRAINT:
identifier : co104
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('18') = 1
definition : flow('17',s)*boreAreaParameter('6')+flow('18',s)*rodAreaParameter('6') = 0;
CONSTRAINT:
identifier : co105
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('19') = 1
definition : flow('17',s)*boreAreaParameter('7')+flow('18',s)*rodAreaParameter('7') = 0;
CONSTRAINT:
identifier : co106
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('13') = 1
definition : flow('17',s)= velocity('19',s)*boreAreaParameter('1');
CONSTRAINT:
identifier : co107
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('14') = 1
definition : flow('17',s)= velocity('19',s)*boreAreaParameter('2');
CONSTRAINT:
identifier : co108
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('15') = 1
definition : flow('17',s)= velocity('19',s)*boreAreaParameter('3');
CONSTRAINT:
identifier : co109
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('16') = 1
definition : flow('17',s)= velocity('19',s)*boreAreaParameter('4');
CONSTRAINT:
identifier : co110
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('17') = 1
definition : flow('17',s)= velocity('19',s)*boreAreaParameter('5');
CONSTRAINT:
identifier : co111
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('18') = 1
definition : flow('17',s)= velocity('19',s)*boreAreaParameter('6');
CONSTRAINT:
identifier : co112
index domain : s
property : IndicatorConstraint
activating condition : cotsVar('19') = 1
definition : flow('17',s)= velocity('19',s)*boreAreaParameter('7');
CONSTRAINT:
identifier : c113
```

index domain : s
definition : pressure('14',s)=pressure('15',s);
CONSTRAINT:
identifier : c114
index domain : s
definition : flow('14',s)=-flow('15',s);
CONSTRAINT:
identifier : c115
index domain : s
definition : forceVariable('67',s)=force('24',s);
CONSTRAINT:
identifier : c116
index domain : s
definition : velocityVariable('68',s)=velocity('24',s);
CONSTRAINT:
identifier : c117
index domain : s
definition : flow('8',s) + flowPAVariable('25',s)+ flowPBVariable('26',s) = 0;
CONSTRAINT:
identifier : c118
index domain : s
definition : flow('9',s) + flowTAVariable('27',s) + flowTBVariable('28',s) = 0;
CONSTRAINT:
identifier : c119
index domain : s
definition : flow('10',s) - flowPAVariable('25',s) - flowTAVariable('27',s)= 0;
CONSTRAINT:
identifier : c120
index domain : s
definition : flow('11',s) - flowPBVariable('26',s) - flowTBVariable('28',s) = 0;
CONSTRAINT:
identifier : c121
index domain : s
property : IndicatorConstraint
activating condition : offVariable('24',s) = 1
definition : flowPAVariable('25',s) = 0;
CONSTRAINT:
identifier : c122
index domain : s
property : IndicatorConstraint
activating condition : offVariable('24',s) = 1
definition : flowTBVariable('28',s) = 0;
CONSTRAINT:
identifier : c123
index domain : s
property : IndicatorConstraint
activating condition : offVariable('24',s) = 1
definition : flowPBVariable('26',s) = 0;
CONSTRAINT:
identifier : c124
index domain : s
property : IndicatorConstraint
activating condition : offVariable('24',s) = 1
definition : flowTAVariable('27',s) = 0;
CONSTRAINT:
identifier : c125

index domain : s
definition : flow('12',s)=-flow('13',s);
CONSTRAINT:
identifier : c126
index domain : s
property : IndicatorConstraint
activating condition : onVariable('22',s) = 1
definition : flow('12',s) = 0;
CONSTRAINT:
identifier : c127
index domain : s
property : IndicatorConstraint
activating condition : offVariable('24',s) = 1
definition : (pressure('12',s)-pressure('13',s)) = 0;
CONSTRAINT:
identifier : c128
index domain : s
property : IndicatorConstraint
activating condition : onVariable('22',s) = 1
definition : (pressure('8',s)-pressure('10',s))= 0;
CONSTRAINT:
identifier : c129
index domain : s
property : IndicatorConstraint
activating condition : onVariable('22',s) = 1
definition : (pressure('11',s)-pressure('9',s)) = 0;
CONSTRAINT:
identifier : c130
index domain : s
property : IndicatorConstraint
activating condition : backVariable('21',s) = 1
definition : (pressure('10',s)-pressure('9',s)) = 0;
CONSTRAINT:
identifier : c131
index domain : s
property : IndicatorConstraint
activating condition : backVariable('21',s) = 1
definition : (pressure('8',s)-pressure('11',s)) = 0;
CONSTRAINT:
identifier : c132
index domain : s
definition : pressureLossVariable('29',s) <= 0;
CONSTRAINT:
identifier : c133
index domain : s
definition : onVariable('22',s) + backVariable('21',s) + offVariable('24',s) = 1;
CONSTRAINT:
identifier : c134
index domain : s
property : IndicatorConstraint
activating condition : backVariable('21',s) = 1
definition : flowPAVariable('25',s) = 0;
CONSTRAINT:
identifier : c135
index domain : s
property : IndicatorConstraint

activating condition : backVariable('21',s) = 1
definition : flowTBVariable('28',s) = 0;
CONSTRAINT:
identifier : c136
index domain : s
property : IndicatorConstraint
activating condition : onVariable('22',s) = 1
definition : flowPBVariable('26',s) = 0;
CONSTRAINT:
identifier : c137
index domain : s
property : IndicatorConstraint
activating condition : onVariable('22',s) = 1
definition : flowTAVariable('27',s) = 0;
CONSTRAINT:
identifier : c138
index domain : s
property : IndicatorConstraint
activating condition : backVariable('21',s) = 1
definition : flow('12',s) = 0;
SET:
identifier : connectorSet
indices : cs
definition : data {'1','2','3','4','5','6','7','8','9','10','11','12','13','14' };
VARIABLE:
identifier : flowVariable
index domain : (cs,s)
range : free;
CONSTRAINT:
identifier : flowC0
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('5') = 1
definition : (angularVelocity('1',s) - angularVelocity('16',s)) = 0;
CONSTRAINT:
identifier : flowC1
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('5') = 0
definition : flowVariable('9', s) = 0;
CONSTRAINT:
identifier : flowC2
index domain : s
definition : torque('1',s) = flowVariable('9',s);
CONSTRAINT:
identifier : flowC3
index domain : s
definition : (pressure('2',s) - pressure('6',s))= 0;
CONSTRAINT:
identifier : flowC4
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('3') = 1
definition : (pressure('2',s) - pressure('13',s)) = 0;
CONSTRAINT:
identifier : flowC5

index domain : s
property : IndicatorConstraint
activating condition : decisionVars('3') = 0
definition : flowVariable('5', s) = 0;
CONSTRAINT:
identifier : flowC6
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('3') = 1
definition : (pressure('2',s) - pressure('14',s)) = 0;
CONSTRAINT:
identifier : flowC7
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('3') = 0
definition : flowVariable('6', s) = 0;
CONSTRAINT:
identifier : flowC8
index domain : s
definition : flow('2',s) = flowVariable('2',s)+ flowVariable('5',s)+ flowVariable('6',s);
CONSTRAINT:
identifier : flowC9
index domain : s
definition : (pressure('3',s) - pressure('5',s))= 0;
CONSTRAINT:
identifier : flowC10
index domain : s
definition : (pressure('3',s) - pressure('7',s))= 0;
CONSTRAINT:
identifier : flowC11
index domain : s
definition : flow('3',s) = flowVariable('1',s)+ flowVariable('3',s);
CONSTRAINT:
identifier : flowC12
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('3') = 1
definition : (pressure('4',s) - pressure('9',s)) = 0;
CONSTRAINT:
identifier : flowC13
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('3') = 0
definition : flowVariable('7', s) = 0;
CONSTRAINT:
identifier : flowC14
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('4') = 1
definition : (pressure('4',s) - pressure('12',s)) = 0;
CONSTRAINT:
identifier : flowC15
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('4') = 0
definition : flowVariable('8', s) = 0;

```
CONSTRAINT:
identifier : flowC16
index domain : s
definition : flow('4',s) = flowVariable('7',s)+ flowVariable('8',s);
CONSTRAINT:
identifier : flowC17
index domain : s
definition : (pressure('5',s) - pressure('3',s))= 0;
CONSTRAINT:
identifier : flowC18
index domain : s
definition : flow('5',s) = - flowVariable('1',s);
CONSTRAINT:
identifier : flowC19
index domain : s
definition : (pressure('6',s) - pressure('2',s))= 0;
CONSTRAINT:
identifier : flowC20
index domain : s
definition : flow('6',s) = - flowVariable('2',s);
CONSTRAINT:
identifier : flowC21
index domain : s
definition : (pressure('7',s) - pressure('3',s))= 0;
CONSTRAINT:
identifier : flowC22
index domain : s
definition : flow('7',s) = - flowVariable('3',s);
CONSTRAINT:
identifier : flowC23
index domain : s
definition : (pressure('8',s) - pressure('15',s))= 0;
CONSTRAINT:
identifier : flowC24
index domain : s
definition : flow('8',s) = flowVariable('4',s);
CONSTRAINT:
identifier : flowC25
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('3') = 1
definition : (pressure('9',s) - pressure('4',s)) = 0;
CONSTRAINT:
identifier : flowC26
index domain : s
definition : flow('9',s) = - flowVariable('7',s);
CONSTRAINT:
identifier : flowC27
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('7') = 1
definition : (pressure('10',s) - pressure('17',s)) = 0;
CONSTRAINT:
identifier : flowC28
index domain : s
property : IndicatorConstraint
```

activating condition : decisionVars('7') = 0
definition : flowVariable('12', s) = 0;
CONSTRAINT:
identifier : flowC29
index domain : s
definition : flow('10',s) = flowVariable('12',s);
CONSTRAINT:
identifier : flowC30
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('7') = 1
definition : (pressure('11',s) - pressure('18',s)) = 0;
CONSTRAINT:
identifier : flowC31
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('7') = 0
definition : flowVariable('13', s) = 0;
CONSTRAINT:
identifier : flowC32
index domain : s
definition : flow('11',s) = flowVariable('13',s);
CONSTRAINT:
identifier : flowC33
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('4') = 1
definition : (pressure('12',s) - pressure('4',s)) = 0;
CONSTRAINT:
identifier : flowC34
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('6') = 1
definition : (pressure('12',s) - pressure('13',s)) = 0;
CONSTRAINT:
identifier : flowC35
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('6') = 0
definition : flowVariable('10', s) = 0;
CONSTRAINT:
identifier : flowC36
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('6') = 1
definition : (pressure('12',s) - pressure('14',s)) = 0;
CONSTRAINT:
identifier : flowC37
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('6') = 0
definition : flowVariable('11', s) = 0;
CONSTRAINT:
identifier : flowC38
index domain : s
definition : flow('12',s) = - flowVariable('8',s)+ flowVariable('10',s)+ flowVariable('11',s);

```
CONSTRAINT:
identifier : flowC39
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('3') = 1
definition : (pressure('13',s) - pressure('2',s)) = 0;
CONSTRAINT:
identifier : flowC40
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('6') = 1
definition : (pressure('13',s) - pressure('12',s)) = 0;
CONSTRAINT:
identifier : flowC41
index domain : s
definition : flow('13',s) = - flowVariable('5',s)- flowVariable('10',s);
CONSTRAINT:
identifier : flowC42
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('3') = 1
definition : (pressure('14',s) - pressure('2',s)) = 0;
CONSTRAINT:
identifier : flowC43
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('6') = 1
definition : (pressure('14',s) - pressure('12',s)) = 0;
CONSTRAINT:
identifier : flowC44
index domain : s
definition : flow('14',s) = - flowVariable('6',s)- flowVariable('11',s);
CONSTRAINT:
identifier : flowC45
index domain : s
definition : (pressure('15',s) - pressure('8',s))= 0;
CONSTRAINT:
identifier : flowC46
index domain : s
definition : flow('15',s) = - flowVariable('4',s);
CONSTRAINT:
identifier : flowC47
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('5') = 1
definition : (angularVelocity('16',s) - angularVelocity('1',s)) = 0;
CONSTRAINT:
identifier : flowC48
index domain : s
definition : torque('16',s) = - flowVariable('9',s);
CONSTRAINT:
identifier : flowC49
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('7') = 1
definition : (pressure('17',s) - pressure('10',s)) = 0;
```

```
CONSTRAINT:
identifier : flowC50
index domain : s
definition : flow('17',s) = - flowVariable('12',s);
CONSTRAINT:
identifier : flowC51
index domain : s
property : IndicatorConstraint
activating condition : decisionVars('7') = 1
definition : (pressure('18',s) - pressure('11',s)) = 0;
CONSTRAINT:
identifier : flowC52
index domain : s
definition : flow('18',s) = - flowVariable('13',s);
CONSTRAINT:
identifier : flowC53
index domain : s
definition : (velocity('19',s) - velocity('24',s))= 0;
CONSTRAINT:
identifier : flowC54
index domain : s
definition : force('19',s) = flowVariable('14',s);
CONSTRAINT:
identifier : flowC55
index domain : s
definition : force('20',s) = 0;
CONSTRAINT:
identifier : flowC56
index domain : s
definition : flow('21',s) = 0;
CONSTRAINT:
identifier : flowC57
index domain : s
definition : flow('22',s) = 0;
CONSTRAINT:
identifier : flowC58
index domain : s
definition : force('23',s) = 0;
CONSTRAINT:
identifier : flowC59
index domain : s
definition : (velocity('24',s) - velocity('19',s))= 0;
CONSTRAINT:
identifier : flowC60
index domain : s
definition : force('24',s) = - flowVariable('14',s);
CONSTRAINT:
identifier : t0
definition : forceVariable('67','1')=0;
CONSTRAINT:
identifier : t1
definition : velocityVariable('68','1')=0;
CONSTRAINT:
identifier : t2
definition : velocityVariable('68','2')>=1;
CONSTRAINT:
```

```
identifier : t3
definition : forceVariable('67','2')>=1000;
MATHEMATICAL PROGRAM:
identifier : MP
objective :
direction : minimize
constraints : AllConstraints
variables : AllVariables ;
ENDSECTION  ;
PROCEDURE
identifier :  MainInitialization
ENDPROCEDURE  ;
PROCEDURE
identifier :  MainExecution
body      :
Solve MP;
ENDPROCEDURE  ;
PROCEDURE
identifier :  MainTermination
body      :
if ( CaseSaveAll( confirm:2 ) = 1 ) then
return 1;
else
return 0;
endif ;
ENDPROCEDURE  ;
ENDMODEL View ;
```

REFERENCES

Agarwal, M., 1999, *Supporting Automated Design Generation: Function Based Shape Grammars and Insightful Optimization*, Thesis.

Agrawal, A., Levendovszky, T., Sprinkle, J., Shi, F., and Karsai, G., 2002, "Generative Programming Via Graph Transformations in the Model-Driven Architecture," *Workshop on Generative Techniques in the Context of Model Driven Architecture, OOPSLA, Nov*, **5**, pp. 184-195.

Akao, Y., 2004, *Quality Function Deployment*, Productivity Press, New York, NY.

Åkesson, J., Ekman, T., and Hedin, G., 2008, "Development of a Modelica Compiler Using Jastadd," *Electronic Notes in Theoretical Computer Science*, **203**(2), pp. 117-131, 10.1016/j.entcs.2008.03.048.

Åkesson, J., Årzén, K. E., Gäfvert, M., Bergdahl, T., and Tummescheit, H., 2010a, "Modeling and Optimization with Optimica and Jmodelica.Org—Languages and Tools for Solving Large-Scale Dynamic Optimization Problems," *Computers & Chemical Engineering*, **34**(11), pp. 1737-1749, 10.1016/j.compchemeng.2009.11.011.

Åkesson, J., Ekman, T., and Hedin, G., 2010b, "Implementation of a Modelica Compiler Using Jastadd Attribute Grammars," *Science of Computer Programming*, **75**(1–2), pp. 21-38, 10.1016/j.scico.2009.07.003.

Alawneh, L., Debbabi, M., Hassaine, F., Jarraya, Y., and Soeanu, A., 2006, "A Unified Approach for Verification and Validation of Systems and Software Engineering Models," *13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, 2006.*, pp. 10-418.

Alber, R., and Rudolph, S., 2002, "On a Grammar-Based Design Language That Supports Automated Design Generation and Creativity," *Proceedings of IFIP WG5.2 Workshop on Knowledge Intensive CAD (KIC-5)*, Malta, Malta.

Amelunxen, C., Konigs, A., Rotschke, T., and Schurr, A., 2006, "Moflon: A Standard-Compliant Metamodeling Framework with Graph Transformations," *Lecture Notes In Computer Science*, **4066**, pp. 361.

Antonsson, E. K., and Cagan, J., 2001, *Formal Engineering Design Synthesis*, Cambridge University Press.

Arrow, K. J., 1963, *Social Choice and Individual Values*, Wiley, New York, NY, USA.

Bajaj, M., Peak, R. S., and Paredis, C. J. J., 2007a, "Knowledge Composition for Efficient Analysis Problem Formulation Part 2: Approach and Analysis Meta-Model," in *ASME 2007 International Design Engineering Technical Conferences*

& Computers and Information in Engineering Conference, Tzou, H. S. and Jalili, N., Eds., ASME, Las Vegas, Nevada, USA.

Bajaj, M., Peak, R. S., and Paredis, C. J. J., 2007b, "Knowledge Composition for Efficient Analysis Problem Formulation Part 1: Motivation and Requirements," in *ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, Tzou, H. S. and Jalili, N., Eds., ASME, Las Vegas, Nevada, USA.

Baker, N. C., and Fenves, S. J., 1990, "Manipulating Shape and Its Function " *Journal of Computing in Civil Engineering*, **4**(4), pp. 221-238.

Baldwin, C. Y., and Clark, K. B., 1999, *Design Rules: Volume 1. The Power of Modularity*, The MIT Press.

Bascaran, E., Bannerot, R. B., and Mistree, F., 1989, "Hierarchical Selection Decision Support Problems in Conceptual Design," *Engineering Optimization*, **14**, pp. 207-238.

Beltrame, T., and Cellier, F. E., 2006, "Quantised State System Simulation in Dymola/Modelica Using the DEVS Formalism," *Proceedings of the 5th International Modelica Conference*, Kral, C. and Haumer, A. eds., Vienna, Austria, pp. 73-82.

Biegler, L., Grossmann, I., and Westerberg, A., 1997, *Systematic Methods for Chemical Process Design*, Prentice Hall, Old Tappan, NJ.

Bisschop, J., and Roelofs, M., 2006, *Aimms Language Reference*, Paragon Decision Technology, Haarlem, The Netherlands.

Blair, C. E., Jeroslow, R. G., and Lowe, J. K., 1986a, "Some Results and Experiments in Programming Techniques for Propositional Reasoning," *Computers and Operations Research*, **13**, pp. 633-645.

Blair, R., 1986b, "Some Results and Experiments in Programming Techniques for Propositional Logic," *Computers & Operations Research*, **13**(5), pp. 633-645.

Bobrow, D. G., 1984, "Qualitative Reasoning About Physical Systems: An Introduction," *Artificial Intelligence*, **24**(1–3), pp. 1-5, 10.1016/0004-3702(84)90036-5.

Bohm, M. R., Stone, R. B., Simpson, T. W., and Steva, E. D., 2008, "Introduction of a Data Schema to Support a Design Repository," *Computer-Aided Design*, **40**(7), pp. 801-811, 10.1016/j.cad.2007.09.003.

Bolognini, F., Seshia, A. A., and Shea, A. K., 2007, "A Computational Design Synthesis Method for MEMS Using COMSOL," *COMSOL Users Conference*.

Bras, B., and Mistree, F., 1993, "Robust Design Using Compromise Decision Support Problems," *Engineering Optimization*, **21**(3), pp. 213-239, 10.1080/03052159308940976.

Brooke, A., Kendrick, D., Meeraus, A., and Raman, R., 1998, *The General Algebraic Modeling System*, GAMS Development Corporation, Washington, DC, USA.

Bryant, C. R., McAdams, D. A., Stone, R. B., Kurtoglu, T., and Campbell, M. I., 2005, "A Computational Technique for Concept Generation," *ASME Conference Proceedings*, **2005**(4742Xa), pp. 267-276.

Buede, D. M., 2000, *The Engineering Design of Systems: Models and Methods*, John Wiley & Sons, Inc., New York.

Cagan, J., Campbell, M., Finger, S., and Tomiyama, T., 2005, "A Framework for Computational Design Synthesis: Model and Applications," *Journal of Computing and Information Science in Engineering*, **5**, pp. 171.

Campbell, M. I., Cagan, J., and Kotovsky, K., 2000, "Agent-Based Synthesis of Electromechanical Design Configurations," *Journal of Mechanical Design*, **122**(1), pp. 61-69, 10.1115/1.533546.

Campbell, M. I., Cagan, J., and Kotovsky, K., 2003, "The a-Design Approach to Managing Automated Design Synthesis," *Research in Engineering Design*, **14**, pp. 12-24.

Castagne, S., Curran, R., and Collopy, P., 2009, "Implementation of Value-Driven Optimisation for the Design of Aircraft Fuselage Panels," *International Journal of Production Economics*, **117**(2), pp. 381-388, 10.1016/j.ijpe.2008.12.005.

Chanron, V., and Lewis, K. E., 2006, "The Dynamics of Decentralized Design Processes: The Issue of Convergence and Its Impact on Decision Making," *Decision Making in Engineering Design*, ASME Press, New York, pp. 281-290.

Chenouard, R., Granvilliers, L., and Soto, R., 2008, "Model-Driven Constraint Programming," in *Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, ACM, Valencia, Spain, pp. 236-246.

Christen, E., and Bakalar, K., 1999, "VHDL-AMS - a Hardware Description Language for Analog and Mixed-Signal Applications," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, **40**(10), pp. 1263-1272.

Conigliaro, R. A., Kerzhner, A. A., and Paredis, C. J. J., 2009, "Model-Based Optimization of a Hydraulic Backhoe Using Multi-Attribute Utility Theory," *SAE International Journal of Materials & Manufacturing*, **2**(1), pp. 298-309.

Creignou, N., Khanna, S., and Sudan, M., 2001, "Complexity Classifications of Boolean Constraint Satisfaction Problems," Society for Industrial and Applied Mathematics, Philadelphia, PA.

Czarnecki, K., and Helsen, S., 2006, "Feature-Based Survey of Model Transformation Approaches," *IBM Systems Journal*, **45**(3), pp. 621-645, 10.1147/sj.453.0621.

da Silva, J. C., 1998, *Expert System Prototype for Hydraulic System Design Focusing on Concurrent Engineering Aspects*, Ph.D. Thesis, Department of Mechanical Engineering, Federal University of Santa Catarina, Florianopolis, Brazil.

da Silva, J. C., and Back, N., 2000, "Shaping the Process of Fluid Power System Design Applying an Expert System," *Research in Engineering Design*, **12**(1), pp. 8-17.

Dauenhauer, G., Aschauer, T., and Pree, W., 2009, "Variability in Automation System Models," *Formal Foundations of Reuse and Domain Engineering*, pp. 116-125.

Domingos, P., Lowd, D., Kok, S., Poon, H., Richardson, M., and Singla, P., 2008a, "Just Add Weights: Markov Logic for the Semantic Web," *Uncertainty Reasoning for the Semantic Web I*, pp. 1-25.

Domingos, P., and Richardson, M., 2008b, "Markov Logic: A Unifying Framework for Statistical Relational Learning."

Donndelinger, J., 2006, "A Decision-Based Perspective on the Vehicle Development Process," *Decision Making in Engineering Design*, American Society of Mechanical Engineers, New York, pp. 217-225.

Dyn, N., and Ron, A., 1995, "Radial Basis Function Approximation: From Gridded Centres to Scattered Centres," *Proceedings of the London Mathematical Society*, **s3-71**(1), pp. 76-108, 10.1112/plms/s3-71.1.76.

Eaton, 1998, *Pump and Motor Sizing Guide*, Eaton Corporation Hydraulics Division, Eden Prarie, MN.

Emery, D., and Hilliard, R., 2009, "Every Architecture Description Needs a Framework: Expressing Architecture Frameworks Using ISO/IEC 42010," *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pp. 31-40.

Emmerich, M., Grotzner, M., and Schutz, M., 2001, "Design of Graph-Based Evolutionary Algorithms: A Case Study for Chemical Process Networks," *Evolutionary Computation*, **9**(3), pp. 329-354.

Eppinger, S. D., Sosa, M. E., and Rowles, C. M., 2000, "Designing Modular and Integrative Systems," *2000 ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Baltimore, Maryland, USA.

Estefan, J., 2007, "Survey of Model-Based Systems Engineering (MBSE) Methodologies," *Incose MBSE Focus Group*, **25**.

Fenves, S., Foufou, S., Bock, C., and Sriram, R. D., 2008, "CPM2: A Core Model for Product Data," *Journal of Computing and Information Science in Engineering*, **8**(1), pp. 014501.

Fischer, T., Niere, J., Torunski, L., and ZÃ¼ndorf, A., 1998, "Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java," *Theory and Application of Graph Transformations*, pp. 157-167.

Fisher, J., 1998, "Model-Based Systems Engineering: A New Paradigm," *INCOSE Insight*, **1**(3), pp. 3-16.

Fourer, R., Gay, D. M., and Kernighan, B. W., 1990a, "A Modeling Language for Mathematical

Programming.," *Management Science*, **36**, pp. 519-554.

Fourer, R., Gay, D. M., and Kernighan, B. W., 1990b, "A Modeling Language for Mathematical Programming," *Management Science*, pp. 519-554.

Friedenthal, S., Moore, A., and Steiner, R., 2008, *A Practical Guide to SysML: The Systems Modeling Language*, Morgan Kaufmann, Waltham, MA, USA.

Friedman, J. H., 1991, "Multivariate Adaptive Regression Splines," *The Annals of Statistics*, **19**(1), pp. 1-141.

Fritzson, P., 2004, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, IEEE Press, Piscataway, NJ, USA.

Gamma, E., 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, New York, NY, USA.

Gano, S., Renaud, J., Martin, J., and Simpson, T., 2006, "Update Strategies for Kriging Models Used in Variable Fidelity Optimization," *Structural and Multidisciplinary Optimization*, **32**(4), pp. 287-298, 10.1007/s00158-006-0025-y.

Garrido, J. M., 2001, *Object-Oriented Discrete-Event Simulation with Java: A Practical Introduction*, Springer US, New York, NY, USA.

Gero, J., 1996, "Creativity, Emergence and Evolution in Design," *Knowledge-Based Systems*, **9**(7), pp. 435-448.

Gershenson, J. K., Prasad, G. J., and Allamneni, S., 1999, "Modular Product Design: A Life-Cycle View," *Journal of Integrated Design & Process Science*, **3**(4), pp. 13-26.

Goldberg, D. E., 1989, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Greenyer, J., Kindler, E., 2007, "Reconciling Tggs with Qvt," in *Model Driven Engineering Languages and Systems, MoDELS 2007*, Springer, Berlin / Heidelberg.

Grosse, I. R., Milton-Benoit, J. M., and Wileden, J. C., 2005, "Ontologies for Supporting Engineering Analysis Models," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **19**(1), pp. 1-18.

Grossmann, I., 2002, "Review of Nonlinear Mixed-Integer and Disjunctive Programming Techniques," *Optimization and Engineering*, **3**(3), pp. 227-252.

Haga, M., Hiroshi, W., and Fujishima, K., 2001, "Digging Control System for Hydraulic Excavator," *Mechatronics*, **11**(6), pp. 665-676, 10.1016/s0957-4158(00)00043-x.

Haq, M., and Rudolph, S., 2005, "A Design Language for Generic Space Frame Structure Design," *1st IFIP TC5 Working Conference on Computer Aided Innovation*, Ulm, Germany, pp. 201-215.

Hazelrigg, G. A., 1998, "A Framework for Decision-Based Engineering Design," *Journal of Mechanical Design*, **120**(4), pp. 653-658.

Hazelrigg, G. A., 2012, *Fundamentals of Decision Making for Engineering Design and Systems Engineering*, George A Hazelrigg, Washington, D.C.

Heinicke, M. U., and Hickman, A., 2000, "Eliminate Bottlenecks with Integrated Analysis Tools in Em-Plant," *2000 Winter Simulation Conference*, IEEE, **1**, pp. 229-231.

Heisserman, J., 1994, "Generative Geometric Design," *IEEE Computer Graphics & Applications*, **14**(2), pp. 37-45.

Helms, B., Shea, K., and Hoisl, F., 2009, "A Framework for Computational Design Synthesis Based on Graph-Grammars and Function-Behavior-Structure," *ASME IDETC*.

Hodges, J., 1992, "Naive Mechanics: A Computational Model of Device Use and Function in Design Improvisation," *IEEE Expert*, **7**(1), pp. 14-27.

Holland, J. H., 1992, *Adaptation in Natural and Artificial Systems*, MIT Press Cambridge, MA, USA.

Horváth, I., Vergeest, J. S. M., and Kuczogi, G., 1998, "Development and Application of Design Concept Ontologies for Contextual Conceptualization," *1998 ASME Design Engineering Technical Conferences*, Atlanta, GA.

Huang, E., Ramamurthy, R., and McGinnis, L. F., 2007, "System and Simulation Modeling Using SysML," in *Proceedings of the 39th conference on Winter Simulation*, Tew, J., Ed., IEEE Press, Washington D.C., pp. 796-803.

Hunt, J. E., Lee, M. H., and Price, C. J., 1993, "Applications of Qualitative Model-Based Reasoning," *Control Engineering Practice*, **1**(2), pp. 253-266, 10.1016/0967-0661(93)91615-4.

Hurwicz, L., 1960, "Optimality and Informational Efficiency in Resource Allocation Processes," *Mathematical methods in the social sciences*, pp. 27-46.

Ingham, M. D., Rasmussen, R. D., Bennett, M. B., and Moncada, A. C., 2006, "Engineering Complex Embedded Systems with State Analysis and the Mission Data System," in *AIAA Intelligent Systems Technical Conference*, Chicago, Illinois.

International Business Machines Corp, 2009, V12. 1: User's Manual for Cplex, ftp://ftp.software.ibm.com/software/websphere/ilog/docs/optimization/cplex/ps_usrmancplex.pdf.

ISO/IEC, 2005, Unified Modeling Language Specification, http://www.omg.org/cgi-bin/apps/doc?formal/05-04-01.pdf.

ISO/IEC, 2007, "ISO/IEC 42010: 2007-Systems and Software Engineering–Recommended Practice for Architectural Description of Software-Intensive Systems," Technical report, International Standardization Organization/International Electrotechnical Commission

Jackson, D., 2002, "Alloy: A Lightweight Object Modelling Notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **11**(2), pp. 256-290.

Jin, R., Du, X., and Chen, W., 2003, "The Use of Metamodeling Techniques for Optimization under Uncertainty," *Structural and Multidisciplinary Optimization*, **25**(2), pp. 99-116, 10.1007/s00158-002-0277-0.

Jobe, J. M., 2008, *Multi-Aspect Component Models: Enabling the Reuse of Engineering Analysis Models in SysML*, Thesis, Georgia Institute of Technology.

Johnson, T., Kerzhner, A., Paredis, C. J. J., and Burkhart, R., 2012, "Integrating Models and Simulations of Continuous Dynamics into SysML," *Journal of Computing and Information Science in Engineering*, **12**(1), pp. 011002.

Johnson, T. A., Paredis, C. J. J., and Burkhart, R., 2008, "Integrating Models and Simulations of Continuous Dynamics into SysML," in *Modelica Conference 2008*, Bielefeld, Germany.

Karandikar, H., Srinivasan, R., Mistree, F., and Fuchs, W. J., 1989, "Compromise: An Effective Approach for the Design of Pressure Vessels Using Composite Materials," *Computers & Structures*, **33**(6), pp. 1465-1477, 10.1016/0045-7949(89)90487-2.

Karban, R., Zamparelli, M., Bauvir, B., Koehler, B., Noethe, L., and Balestra, A., 2008, "Exploring Model Based Engineering for Large Telescopes-Getting Started with Descriptive Models," **7017**, pp. 44.

Keeney, R. L., and Raiffa, H., 1976, *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*, Jon Wiley and Sons, New York, NY, USA.

Kelton, W. D., Sadowski, R. P., and Sadowski, D. A., 2002, *Simulation with ARENA*, McGraw-Hill, New York, NY.

Kerzhner, A. A., and Paredis, C. J. J., 2009, "Using Domain Specific Languages to Capture Design Synthesis Knowledge for Model-Based Systems Engineering," *ASME Conference Proceedings*, **2009**(48999), pp. 1399-1409.

Kerzhner, A. A., and Paredis, C. J. J., 2010, "Model-Based System Verification: A Formal Framework for Relating Analyses, Requirements, and Tests," *Proceedings of the 4th International Workshop on Multi-Paradigm Modeling - MPM'10*, Oslo, Norway.

Kerzhner, A. A., Jobe, J. M., and Paredis, C. J. J., 2011, "A Formal Framework for Capturing Knowledge to Transform Structural Models into Analysis Models," *Journal of Simulation*, **5**(3), pp. 202-216.

Königs, A., and Schürr, A., 2006, "Tool Integration with Triple Graph Grammars - a Survey," *Electronic Notes in Theoretical Computer Science*, **148**(1), pp. 113-150, 10.1016/j.entcs.2005.12.015.

Kopena, J. B., and Regli, W. C., 2003, "Functional Modeling of Engineering Designs for the Semantic Web," *Data Engineering*, **26**(4), pp. 55-61.

Koza, J., 2001, "Automatic Synthesis of Both the Topology and Numerical Parameters for Complex Structures Using Genetic Programming," *Engineering Design Synthesis: Understanding, Approaches, and Tools*, Springer-Verlag London, pp.

Koza, J., 2010, "Human-Competitive Results Produced by Genetic Programming," *Genetic Programming and Evolvable Machines*, **11**(3), pp. 251-284, 10.1007/s10710-010-9112-3.

Kumar, V., 1992, "Algorithms for Constraint-Satisfaction Problems: A Survey," *AI magazine*, **13**(1), pp. 32.

Lastusilta, T., Bussieck, M. R., and Westerlund, T., 2007, "Comparison of Some High-Performance MINLP Solvers," *Chemical Engineering Transactions*, **11**.

Le Metayer, D., 1998, "Describing Software Architecture Styles Using Graph Grammars," *IEEE Transactions on Software Engineering*, **24**(7), pp. 521-533.

Lee, J., and Leyffer, S., 2011, *Mixed Integer Nonlinear Programming*, Springer Verlag, New York, NY, USA.

Leweling, K.-U., and Stein, B., 2000, "Hybrid Constraints in Automated Model Synthesis and Model Processing," *Electronic Notes in Discrete Mathematics*, **4**(0), pp. 68, 10.1016/s1571-0653(05)80107-6.

Maier, M., and Rechtin, E., 2000, *The Art of Systems Architecting*, CRC Press, Boca Raton, Florida.

Malak, R. J., 2008, *Using Parameterized Efficient Sets to Model Alternatives for Systems Design Decisions*, Thesis, Mechanical Engineering, Georgia Institute of Technology, Atlanta, GA.

Malak, R. J., and Paredis, C. J. J., 2010, "Using Parameterized Pareto Sets to Model Design Concepts," *Journal of Mechanical Design*, **132**(4), pp. 041007-041011.

Matheron, G., 1963, "Principles of Geostatistics," *Economic Geology*, **58**(8), pp. 1246-1266, 10.2113/gsecongeo.58.8.1246.

Mathews, J. H., and Fink, K. D., 1998, *Numerical Methods Using Matlab*, Simon & Schuster.

Mathworks, S., 2008, "Simulink," vol. 2008.

Mattsson, S. E., and Elmqvist, H., 1998, "An Overview of the Modeling Language Modelica," *Eurosim '98 Simulation congress*, Helsinki, Finland.

McCandlish, D., and Dorey, R. E., 1984, "The Mathematical Modelling of Hydrostatic Pumps and Motors," *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, **198**(3), pp. 165-174, 10.1243/pime_proc_1984_198_062_02.

Min, B. I., Kerzhner, A. A., and Paredis, C. J. J., 2011, "Process Integration and Design Optimization for Model-Based Systems Engineering with SysML," in *the ASME 2011 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (IDETC/CIE2011)*, Washington, D.C., USA.

Mocko, G., Malak Jr., R. J., Paredis, C. J. J., and Peak, R., 2004, "A Knowledge Repository for Behavioral Models in Engineering Design," *ASME Computers and Information in Engineering Conference*, Salt Lake City, UT.

Modelica Association, 2005, "Modelica Language Specification," Linköping, Sweden.

Modelica Association, 2012, Modelica and the Modelica Association http://www.modelica.org/

Moore, R., Romero, D., and Paredis, C., 2011, "A Rational Design Approach to Gaussian Process Modeling for Variable Fidelity Models," in *Proceedings of the ASME 2011 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (IDETC/CIE2011)*, Washington, D.C., USA.

Mullins, S., and Rinderle, J. R., 1991a, "Grammatical Approaches to Engineering Design, Part I: An Introduction and Commentary," *Research in Engineering Design*, **2**(3), pp. 121-135.

Mullins, S., and Rinderle, J. R., 1991b, "Grammatical Approaches to Engineering Design, Part I: An Introduction and Commentary," *Research in Engineering Design*, **2**, pp. 121-153.

Nagl, M., 1979, *Graph-Grammatiken, Theorie, Implementierung, Anwendungen*, Vieweg, Braunschweig, DE.

No Magic Inc., 2012, Magicdraw UML, http://www.magicdraw.com.

Object Management Group, 2006, OMG Systems Modeling Language (OMG SysML), V1.0, http://www.omgsysml.org/.

Object Management Group, 2007, Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, http://www.omg.org/docs/formal/08-04-03.pdf.

Object Management Group, 2010, SysML-Modelica (SyM) 1.0 - Beta 1, http://www.omg.org/spec/SyM/1.0/Beta1/.

Olewnik, A., and Lewis, K., 2006, "A Decision Support Framework for Flexible System Design," *Journal of Engineering Design*, **17**(1), pp. 75-97, 10.1080/09544820500274019.

Pahl, G., Beitz, W., Feldhunen, J., and Grote, K.H., 2007, *Engineering Design: A Systematic Approach*, Springer, London, UK.

Paredis, C., Bernard, Y., Burkhart, R. M., de Koning, H. P., Friedenthal, S., Fritzson, P., Rouquette, N. F., and Schamai, W., 2010, "An Overview of the SysML-Modelica Transformation Specification," *Proceedings of the 2010 INCOSE Symposium*, Chicago, IL, USA, pp. 1-14.

Paredis, C. J. J., Diaz-Calderon, A., Sinha, R., and Khosla, P. K., 2001, "Composable Models for Simulation-Based Design," *Engineering with Computers*, **17**(2), pp. 112-128.

Paredis, C. J. J., and Johnson, T., 2008, "Using Omg's SysML to Support Simulation," in *Proceedings of the 40th Conference on Winter Simulation*, Mason, S., Hill, R., Mönch, L., and Rose, O., Eds., Winter Simulation Conference, Miami, Florida, pp. 2350-2352.

Parker, 2002, *Design Engineers Handbook*, Parker Hannifin Corporation, Cleveland, OH.

Parnell, G. S., Driscoll, P. J., and Henderson, D. L., 2011, *Decision Making in Systems Engineering and Management*, John WIley & Sons, Inc., Hoboken, New Jersey.

Peak, R. S., Fulton, R. E., Nishigaki, I., and Okamoto, N., 1998, "Integrating Engineering Design and Analysis Using a Multi-Representation Approach," *Engineering with Computers*, **14**(2), pp. 93-114.

Peak, R. S., Burkhart, R. M., Friedenthal, S. A., Wilson, M. W., Bajaj, M., and Kim, I., 2007, "Simulation-Based Design Using SysML-Part1: A Parametrics Primer," in *INCOSE Intl. Symposium*, San Diego, CA.

Pedersen, H. C., 2007, *Automated Hydraulic System Design and Power Management in Mobile Applications*, Institut for Energiteknik, Aalborg Universitet, Copenhagen, DK.

Pugh, S., 1990, *Total Design - Integrated Methods for Successful Product Engineering*, Addison-Wesley Publishing Company, Wokingham, England.

Pugh, S., 1991, *Total Design: Integrated Methods for Successful Product Engineering*, Addison-Wesley Pub. Co., Reading, MA.

Qamar, A., During, C., and Wikander, J., 2009, "Designing Mechatronic Systems, a Model-Based Perspective, an Attempt to Achieve SysML-Matlab/Simulink Model Integration," *IEEE/ASME International Conference on Advanced Intelligent Mechatronics, 2009. AIM 2009.*, pp. 1306-1311.

Rachuri, S., Baysal, M. M., Roy, U., FouFou, S., Bock, C., Fenves, S., Subrahmanian, E., Lyons, K., and Sriram, R. D., 2005, "Information Models for Product Representation: Core and Assembly Models," *International Journal of Product Development*, **2**(3), pp. 207-235.

Rinderle, J. R., 1991, "Grammatical Approaches to Engineering Design, Part Ii: Melding Configuration and Parametric Design Using Attribute Grammars," *Research in Engineering Design*, **2**, pp. 137-146.

Saaty, T. L., 1990, "How to Make a Decision: The Analytic Hierarchy Process," *European Journal of Operational Research*, **48**(1), pp. 9-26, 10.1016/0377-2217(90)90057-i.

Sage, A. P., and Armstrong, J. E., Jr., 2000a, *Introduction to Systems Engineering*, John Wiley & Sons, New York, NY.

Sage, A. P., and Armstrong Jr., J. E., 2000b, *Introduction to Systems Engineering*, John Wiley & Sons, New York, NY.

Sager, S., 2012, "A Benchmark Library of Mixed-Integer Optimal Control Problems

Mixed Integer Nonlinear Programming," Springer New York, **154**, pp. 631-670.

Sahinidis, N. V., 1996, "Baron: A General Purpose Global Optimization Software Package," *Journal of Global Optimization*, **8**(2), pp. 201-205.

Sasajima, M., Kitamura, Y., Ikeda, M., and Mizoguchi, R., 1995, "Fbrl: A Function and Behavior Representation Language," *Proc. of IJCAI*, **95**, pp. 1830-1836.

Sauer-Sunstrand, 1997, *Selection of Driveline Components*, Sauer-Sunstrand Company, Ames, IA.

Saxena, T., and Karsai, G., 2010, "Mde-Based Approach for Generalizing Design Space Exploration," *Model Driven Engineering Languages and Systems*, Springer Berlin / Heidelberg, **6394**, pp. 46-60.

Schmidt, L. C., and Cagan, J., 1996, "Configuration Design: An Integrated Approach Using Grammars," *Proceedings of the 8th ASME International Conference on Design Theory and Methodology*, Irvine, CA.

Schmidt, L. C., and Cagan, J., 1997, "GGREADA: A Graph Grammar-Based Machine Design Algorithm," *Research in Engineering Design*, **9**(4), pp. 195-213.

Schmidt, L. C., and Cagan, J., 1998, "Optimal Configuration Design: An Integrated Approach Using Grammars," *ASME Journal of Mechanical Design*, **120**(1), pp. 2-9.

Schürr, A., 1994, "Specification of Graph Translators with Triple Graph Grammars," in *WG'94 Workshop on Graph-Theoretic Concepts in Computer Science*.

Shah, A., Kerzhner, A., Schaefer, D., and Paredis, C., 2010a, "Multi-View Modeling to Support Embedded Systems Engineering in SysML

Graph Transformations and Model-Driven Engineering," Springer Berlin / Heidelberg, **5765**, pp. 580-601.

Shah, A. A., 2010b, *Combining Mathematical Programming and SysML for Component Sizing as Applied to Hydraulic Systems*, Masters Thesis, Mechanical Engineering, Georgia Institute of Technology, Atlanta.

Shah, A. A., Paredis, C. J. J., Burkhart, R., and Schaefer, D., 2010c, "Combining Mathematical Programming and SysML for Automated Component Sizing of Hydraulic Systems," *ASME Conference Proceedings*, **2010**(44113), pp. 1231-1245.

Shupe, J. A., Mistree, F., and Sobieszanski-Sobieski, J., 1987, "Compromise: An Effective Approach for the Hierarchical Design of Structural Systems," *Computers & Structures*, **26**(6), pp. 1027-1037, 10.1016/0045-7949(87)90119-2.

Simmetrix Inc., 2006, Simulation Application Suite, http://simmetrix.com/products/SimulationApplicationSuite/main.html.

Simpson, T. W., Mauery, T. M., Korte, J. J., and Mistree, F., 2001, "Kriging Models for Global Approximation in Simulation-Based Multidisciplinary Design Optimization," *AIAA Journal*, **39**(12), pp. 2233-2241.

Simulink (The Mathworks), 2008, Simulink, http://www.mathworks.com/products/simulink/.

Smullyan, R., 1995, *First-Order Logic*, Dover Publications.

Sobek Ii, D. K., Ward, A. C., and Liker, J. K., 1999, "Toyota's Principles of Set-Based Concurrent Engineering," *Sloan Management Review*, **40**(2), pp. 67-83.

Sobieszczanski-Sobieski, J., and Haftka, R. T., 1997, "Multidisciplinary Aerospace Design Optimization: Survey of Recent Developments," *Structural and Multidisciplinary Optimization*, **14**(1), pp. 1-23, 10.1007/bf01197554.

Stahl, T., Voelter, M., and Czarnecki, K., 2006, *Model-Driven Software Development: Technology, Engineering, Management*, John Wiley & Sons.

Starling, A. C., and Shea, K., 2005, "A Parallel Grammar for Simulation-Driven Mechanical Design Synthesis," *ASME Conference Proceedings*, **2005**(4739X), pp. 427-436.

Stiny, G., 1980, "Introduction to Shape and Shape Grammars," *Environment and Planning B*, **7**, pp. 343-351.

Stone, R. B., and Wood, K. L., 2000, "Development of a Functional Basis for Design," *Journal of Mechanical Design*, **122**, pp. 359-370.

Summers, J. D., and Shah, J. J., 2010, "Mechanical Engineering Design Complexity Metrics: Size, Coupling, and Solvability," *Journal of Mechanical Design*, **132**(2), pp. 021004-021011.

Szykman, S., Sriram, R., Bochenek, C., and Racz, J., 1998, "The Nist Design Repository Project," *Advances in Soft Computing - Engineering Design and Manufacturing*, Roy, R., Furuhashi, T., and Chawdhry, P. K. eds., Springer-Verlag, London, pp. 5-19.

Tawarmalani, M., and Sahinidis, N. V., 2004, "Global Optimization of Mixed-Integer Nonlinear Programs: A Theoretical and Computational Study," *Mathematical Programming*, **99**(3), pp. 563-591.

Thompson, S. C., and Paredis, C. J. J., 2010, "An Investigation into the Decision Analysis of Design Process Decisions," *Journal of Mechanical Design*, **132**(12), pp. 121009-121009.

Thurston, D. L., 1991, "A Formal Method for Subjective Design Evaluation with Multiple Attributes," *Research in Engineering Design*, **3**(2), pp. 105-122, 10.1007/bf01581343.

Trujillo, S., Garate, J., Lopez-Herrejon, R., Mendialdua, X., Rosado, A., Egyed, A., Krueger, C., and De Sosa, J., 2010, "Coping with Variability in Model-Based Systems Engineering: An Experience in Green Energy," *Modelling Foundations and Applications*, pp. 293-304.

Tzilla, E., Robert, E. F., and Atef, B., 2001, "Aspect-Oriented Programming: Introduction," *Communications of The ACM*, **44**(10), pp. 29-32.

Ulrich, K., and Tung, K., 1991, "Fundamentals of Product Modularity," *1991 ASME Design Technical Conferences - Conference on Design / Manufacture Integration*, Miami, Florida.

Umeda, Y., Takeda, H., Tomiyama, T., and Yoshikawa, H., 1990, "Function, Behavior, and Structure," *Applications of Artificial Intelligence in Engineering V*, Springer-Verlag, Berlin, Germany, **1**, pp. 177-193.

Varga, A., and Hornig, R., 2008, "An Overview of the Omnet++ Simulation Environment," ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 1-10.

von Neumann, J., and Morgenstern, O., 1980, *Theory of Games and Economic Behavior*, Princeton University Press, Princeton, NJ.

Vong, C., Leung, T., and Wong, P., 2002, "Case-Based Reasoning and Adaptation in Hydraulic Production Machine Design," *Engineering Applications of Artificial Intelligence*, **15**(6), pp. 567-585.

Wallace, D., Pahng, G. D. F., and Bae, S., 1998, "Web-Based Collaborative Design Modeling and Decision Support," *1998 ASME Design Engineering Technical Conferences and Engineering in Information Management Conference*, Atlanta, Georgia, USA.

Wang, G. G., and Shan, S., 2007, "Review of Metamodeling Techniques in Support of Engineering Design Optimization," *Journal of Mechanical Design*, **129**(4), pp. 370-380.

Warmer, J., and Kleppe, A., 2003, *The Object Constraint Language: Getting Your Models Ready for MDA*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Weisemoller, I., and Schurr, A., 2007, "A Comparison of Standard Compliant Ways to Define Domain Specific Languages," *Proceedings of the 4th International Workshop on Software Language Engineering*, Nashville, TN, USA, pp. 31-45.

Westman, R., Sargent, C., and Burton, R., 1987, "A Knowledge-Based Modular Approach to Hydraulic Circuit Design," *Computers in Engineering*, **1**, pp. 37-41.

Williams, H. P., 1999, *Model Building in Mathematical Programming, 4th Edition*, John Wiley & Sons Inc., Hoboken, NJ.

Wyatt, D., Wynn, D., Jarrett, J., and Clarkson, P., 2012, "Supporting Product Architecture Design Using Computational Design Synthesis with Network Structure Constraints," *Research in Engineering Design*, **23**(1), pp. 17-52, 10.1007/s00163-011-0112-y.

Yeomans, H., and Grossmann, I., 1999, "A Systematic Modeling Framework of Superstructure Optimization in Process Synthesis," *Computers & Chemical Engineering*, **23**(6), pp. 709-731.

Yunes, T., Aron, I. D., and Hooker, J. N., 2010, "An Integrated Solver for Optimization Problems," *Oper. Res.*, **58**(2), pp. 342-356, 10.1287/opre.1090.0733.

Zeigler, B. P., 1987, "Hierarchical, Modular Discrete-Event Modelling in an Object-Oriented Environment," *Simulation*, **49**(5), pp. 219.

Zeigler, B. P., Ball, G., Cho, H., Lee, J., and Sarjoughian, H., 1999, "Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions," in *Fall Simulation Interoperability Workshop*, Orlando, FL.