USING DOMAIN SPECIFIC LANGUAGES TO CAPTURE DESIGN KNOWLEDGE FOR MODEL-BASED SYSTEMS ENGINEERING

A Thesis
Presented to
The Academic Faculty

by

Aleksandr A. Kerzhner

In Partial Fulfillment
of the Requirements for the Degree of
Master of Science in the
School of Mechanical Engineering

Georgia Institute of Technology
May, 2009

USING DOMAIN SPECIFIC LANGUAGES TO CAPTURE DESIGN KNOWLEDGE FOR MODEL-BASED SYSTEMS ENGINEERING

Approved by:

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

Dr. Leon McGinnis
School of Industrial Engineering
Georgia Institute of Technology

Dr. Dirk Schaefer
School of Mechanical Engineering
Georgia Institute of Technology

Date Approved: March 18, 2009

ACKNOWLEDGEMENTS

Few achievements come without the support of others; and I wish I could completely capture the numerous contributions others have made to the completion of this thesis. I know that I would not have gotten to this point without the invaluable help of many different people.

I first must thank my family for playing a major role in my life, for always being supportive through good times and bad. I'd especially like to thank my mom, Inna, for her instilling in me the importance of a quality education along with a strong work ethic.

Academically speaking, I deeply wish to thank my advisor, Dr. Chris Paredis. He has always been down to earth and willing to provide assistance; as well as being both open minded and critical. Even though he has been in Europe for much of the writing of this thesis, he has found time to provide timely advice. I would also like to thank my reading committee: Dr. Leon McGinnis for always providing challenging (though "simple") questions and Dr. Dirk Schaefer for his support and insight.

I also owe gratitude to my academic family in the Systems Realization Lab. I would like to first thank Richard Malak for the valuable discussions and insightful comments on a regular basis. I'd also like to thank Stephanie Thompson and Roxanne Moore for their support, friendship, and insightful comments. Although I only knew both briefly, Tommy Johnson and Jonathan Jobe provided meaningful discussion as well as some of the ideas used throughout this thesis. Several other members of the SRL stand out as being particularly helpful: John Reap, Aditya Shah, Benjamin Lee, Kevin Davies, and J. Bankston.

This work is funded in part by the Engineering Research Center for Compact and Efficient Fluid Power, supported by the National Science Foundation under Grant No. EEC-0540834. Additional funding was provided by Lockheed Martin. I would also like to thank Andy Schürr for providing access and support for MOFLON and Roger Burkhart at Deere & Company along with Sanford Friedenthal at Lockheed Martin for discussions that helped crystallize the ideas presented in this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

Page

SUMMARY

Design synthesis is a fundamental engineering task that involves the creation of structure from a desired functional specification; it involves both creating a system topology as well as sizing the system's components. Although the use of computer tools is common throughout the design process, design synthesis is often a task left to the designer. At the synthesis stage of the design process, designers have an extensive choice of design alternatives that need to be considered and evaluated.

Designers can benefit from computational synthesis methods in the creative phase of the design process. Recent increases in computational power allow automated synthesis methods for rapidly generating a large number of design solutions. Combining an automated synthesis method with an evaluation framework allows for a more thorough exploration of the design space as well as for a reduction of the time and cost needed to design a system. To facilitate computational synthesis, knowledge about feasible system configurations must be captured. Since it is difficult to capture such synthesis knowledge about any possible system, a design domain must be chosen. In this thesis, the design domain is hydraulic systems.

In this thesis, Model-Driven Software Development concepts are leveraged to create a framework to automate the synthesis of hydraulic systems will be presented and demonstrated. This includes the presentation of a domain specific language to describe the function and structure of hydraulic systems as well as a framework for synthesizing hydraulic systems using graph grammars to generate system topologies. Also, a method using graph grammars for generating analysis models from the described structural system representations is presented. This approach fits in the context of Model-Based Systems Engineering where a variety of formal models are used to represent knowledge about a system. It uses the Systems Modeling Language developed by The Object Management Group (OMG SysML™) as a unifying language for model definition.

CHAPTER 1

INTRODUCTION

Engineered systems are a key component of everyday life from automobiles and aircraft to seemingly ubiquitous electronic devices. Modern systems and therefore modern systems engineering problems are becoming increasingly complex because they often involve the integration of multiple engineering domains, are constrained by often competing objectives, include a multitude of stakeholders, and are inundated by large quantities of design information [54]. Therefore, problems that are often encountered during the system development process are generally the result of poor organization and communication of information or poor management of problem complexity rather than the direct technological concerns that affect individual subsystems. The presence of multiple stakeholders also dictates that design knowledge be explicitly captured to reduce the opportunity for miscommunication.

1.1 Managing complexity with Model-Based Systems Engineering

Some of the complexity can be managed through the formal representation of all aspects of the system engineering problem which has begun with the adoption of a Model-Based Systems Engineering (MBSE) [15]. In MBSE, engineers formally capture knowledge about all aspects of a systems engineering problem in models. There is a plethora of design knowledge that needs to be captured for MBSE. In this thesis, synthesis knowledge is explored independent of analysis knowledge. Synthesis knowledge is the knowledge concerning the generation of design alternatives that are contained within a specific design space while analysis knowledge describes how the behavior of design alternatives can be analyzed.

In support of MBSE, the Object Management Group (OMG) has developed the Systems Modeling Language (SysML) [65] to be a general-purpose systems modeling language that enables systems engineers to create and manage models of engineering systems using well-defined, visual constructs. The formal capture of knowledge using a model-based approach does have disadvantages: there is a higher level of expertise and effort required to explicitly capture knowledge that would otherwise be assumed implicitly.

1.2 Research Questions

To facilitate the use of MBSE, the motivating question becomes:

The Motivating Question:

How can design knowledge be captured such that it can be used effectively and efficiently?

The shift in current industry practice suggests that this question can partially be answered through the adoption of model-based design approaches in a shift away from document-centric design. This shift is embodied in MBSE where models are used throughout the design process to capture knowledge. It is important for models to be defined unambiguously and precisely so they can be easily understood by the various stakeholders involved throughout the design process. Also, these models need to be reusable so that the increase in overall cost associated with formal modeling can be partially mitigated. The MBSE approach requires the development of many different design and analysis models. The question then becomes:

How should these models be defined so that they are unambiguous, reusable, and precise?

To answer this question, this thesis leverages concepts from the domain of computer science, specifically the fields of Model-Driven Architecture (MDA) or Model-Driven Software Development (MDSD). Both involve the use of formal models throughout the design and implementation of software solutions and it seems likely that concepts from these fields can be applied to the domain of systems engineering. Systems engineering shares several characteristics with software engineering most notably the complex interactions between various components. The use of MDSD concepts can simplify the definition and application of models by reducing the need to create problem-specific code for a variety of applications that are discussed throughout the thesis. Instead, computer-aided software engineering (CASE) tools, such as MOFLON [2], are used to generate this code improving the ease of implementing languages with which models can be defined and transformed.

The hypothesis that follows from this argument is:

Design knowledge should be effectively and efficiently captured through the application of Model-Based Software Design concepts such as formal domain specific languages (DSLs) and model transformations.

Although validating this hypothesis is the central motivation of this thesis, it is too broad to be fully addressed. Instead, this thesis takes the first step in the validation process by attempting to confirm a less expansive hypothesis:

Hypothesis:

It is feasible for design knowledge to be captured using DSLs, graph grammars, and other concepts from MDSD.

This hypothesis is still too broad to tackle directly because there is a wide variety of design knowledge that is present throughout the design process, and therefore a wide variety of possible design knowledge to capture. Instead, in this thesis, design knowledge is decomposed into three distinct categories to facilitate exploration of the problem. In this thesis, the hypothesis is explored by decomposition into these three sub-questions:

Question 1:

Is it feasible to capture synthesis knowledge using DSLs and graph grammars to represent and generate design alternatives?

Question 2:

Is it feasible to capture the analysis knowledge needed to generate analysis models from representations of design alternatives using DSLs and graph grammars?

Question 3: Is it feasible to capture the analysis knowledge needed to create simulation models from analysis models using DSLs and graph grammars?

To attempt to answer these questions, a framework that relies on the definition of several domain specific languages (DSLs) through the use of metamodels and model libraries to capture the design knowledge about a particular domain. A DSL is a language that is tailored to describe a particular problem domain. In this framework, the DSL is augmented by the specification of graph-based model transformations designed to *transform* between different models present throughout the systems design process. 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. The use of graph-based transformations also has the advantage of being easily visualized. These advantages will also be thoroughly explored throughout the thesis.

1.3 A DSL-based approach for capturing design knowledge

A variety of different types of knowledge need to be represented in design and systems engineering: requirements and objectives, functions and functional decompositions, logical architectures, physical architectures, behavior, test-cases, allocations, etc. Using formal models to capture all this information and knowledge about analysis is at the foundation of MBSE. Although general purpose modeling languages such as SysML have been defined to capture such systems engineering knowledge, we argue in this thesis that it is often convenient (and maybe more effective) to express this knowledge in a DSL when working in a specific domain. To facilitate integration between the DSLs, SysML (with domain-specific profiles) could be used as an integration framework.

In addition, as is illustrated in Figure 1.1, the concept proposed here is extend this notion of model-based engineering to include also the *transformations* that occur between the different types of models. These transformations incorporate the process knowledge that is needed to solve design problems effectively. The transformations themselves can again be modeled, leading us to the notion of "Model Everything!" [17].

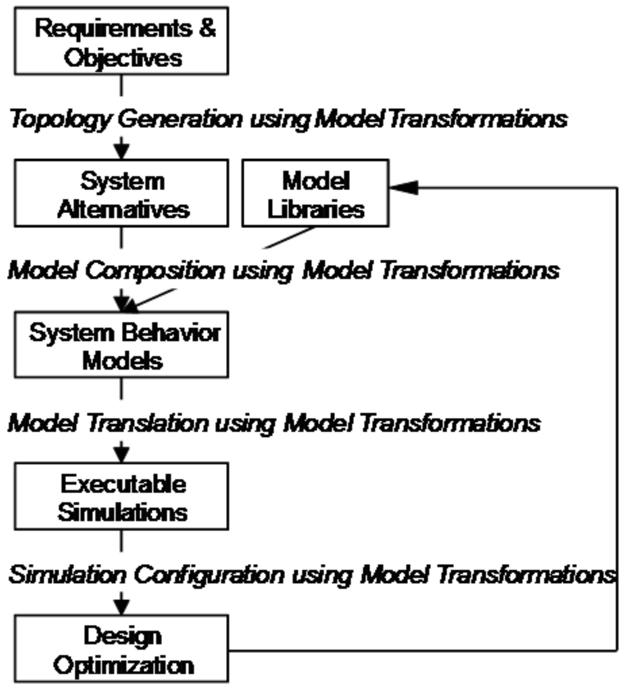

Requirements & Objectives
Topology Generation using Model Transformations
System Alternatives
Model Libraries
Model Composition using Model Transformations
System Behavior Models
Model Translation using Model Transformations
Executable Simulations
Simulation Configuration using Model Transformations
Design Optimization

Figure 1.1: Model Everything! — both representations and transformations.

In this thesis, an effort is presented towards capturing design knowledge through the use of DSLs. Formal models are used to capture knowledge about the space of system design alternatives. This includes the formal representation of these alternatives as well as model or graph transformations for generating instances within this design space.

In addition to the design space, one should define spaces in which the design problem itself is defined (i.e., objectives, requirements, context, etc.), as well as spaces in which the system is described from different viewpoints — functional, behavioral, at different levels of abstraction, from different disciplinary perspectives, etc. An important part of the overall vision for this research is that these different views are formally related to each other through models or model transformations so that the views can efficiently be updated and kept consistent.

1.4 SysML as a Unifying Language

Currently, system engineering problems are solved using a wide range of domain-specific modeling languages and modeling tools. Unfortunately, these domain-specific modeling languages are often implicitly defined. Moreover, it is unlikely that a single unified modeling language will be able to model in sufficient detail the large number of system aspects addressed by current domain-specific languages. One should not "reinvent the wheel" by creating an all-encompassing systems engineering language capable of modeling and simulating every aspect of a system. [28] On the other hand, managing a large number of models in different languages also poses problems, including communication ambiguity and the preservation of information consistency. To alleviate these problems, formal and precise definitions of these domain-specific modeling languages are needed to allow for the integration of these languages.

SysML can provide a foundation for this model integration because of its well-defined yet general constructs which can be easily linked together. SysML can also be used as a foundation for the definition of formal languages that a modeler can use to describe the interaction between system models. There is the additional advantage of easy integration of models in described in these formal languages with the capabilities provided by SysML modeling tools such as the visual and formal modeling of requirements and system behavior.

1.5 Hydraulic systems as a representative example

Since the presented approach relies on the definition of domain specific languages, it is advantageous to choose a domain representative of complex engineered systems. The domain chosen here is one of hydraulic systems. From a systems engineering perspective, hydraulic systems have the interesting characteristic that they are circuit-like; that is, they consist of discrete components that are configured or composed into complex systems. This modularity in the physical system has been introduced to facilitate their design and manufacture. Modular components not only provide economies of scale in the manufacturing process, but they also simplify the design problem by decomposing the system into functional units that have simple and clearly defined interfaces through which they interact with each other. The hydraulic circuits investigated throughout the thesis are similar to common hydraulic circuits found in a wide range of application but especially common in off-road construction equipment such as excavators or backhoes.

1.6 Thesis Structure

According to the hypothesis and related questions, the objective of the work presented is to apply formal domain-specific language and graph grammar concepts to capture knowledge during three distinct transformations commonly present in the design process. Before delving into the validation of this hypothesis and the answering of the related questions, an overview of related work is covered in Chapter 2. An overview of important MDSD concepts is provided in Section 2.1. Standard approaches from MDSD to performing graph transformations and defining DSLs in Sections 2.2 and 2.3 respectively. Some relevant constructs from SysML are covered in Section 2.4.

The next three chapters have similar structure: each investigates the answer to one of the related questions. To simplify the presentation, work related specifically to the question being addressed is contained within each chapter. In Chapter 3, an attempt is made to answer Question 1 by providing a framework for capturing synthesis knowledge. In Chapter 4, Question 2 is addressed by building on work by Jobe that captures analysis knowledge in reusable containers called Multi-Aspect Component Models [25]. In Chapter 5, Question 3 is addressed by demonstrating the approach on the generation of Modelica continuous dynamics simulation models from the analysis models created in Chapter 4. Modelica is used as a representative example of various simulation languages. Chapter 5 builds strongly on the work presented by Johnson [27] where SysML models are transformed into Modelica models. Johnson's work is extended by applying MDSD techniques to defining an explicit DSL for the simulation models and to defining and implementing the transformations.

1.7 Prelimanary Reading

For readers unfamiliar with MDSD and similar topics, there are a number of excellent seminal works in the area. For an overview of MDA concepts, an overview by Mellor et al [34] is recommended. Many core concepts are shared between MDSD and MDA, but for a complete guide to MDSD, work by Stahl et al [62] is an excellent resource. For a brief review of different methods to model and execute model transformations, work by Czarnecki et al [9] is recommended. A glossary of commonly used terms can be found in Appendix A.

CHAPTER 2

RELATED WORK

This chapter covers some high-level concepts as well as related work that is applicable throughout the thesis.

2.1 Common Model Driven Software Development Concepts

Since MDSD concepts are the foundation of the work presented in this thesis, some relevant constructs are presented here. A more thorough examination of all the common concepts can be found in Chapter 4 of Stahl et al [62].

2.1.1 The Domain

The starting point in MDSD is always a *domain*, a “bounded field of interest or knowledge” [62].

2.1.2 Metamodels

Metamodels capture an ontology for the domain, that is the constructs and relationships present independent of any particular independent representation or encoding. Metamodels are used in MDSD to describe the structure of the domain formally. [62] The metamodel defines the *abstract syntax* of the domain and is an instance of a *meta-metamodel*. Metamodels are specified using metamodeling languages. The relationship between models, metamodels, and metamodeling languages is shown in Figure 2.1.

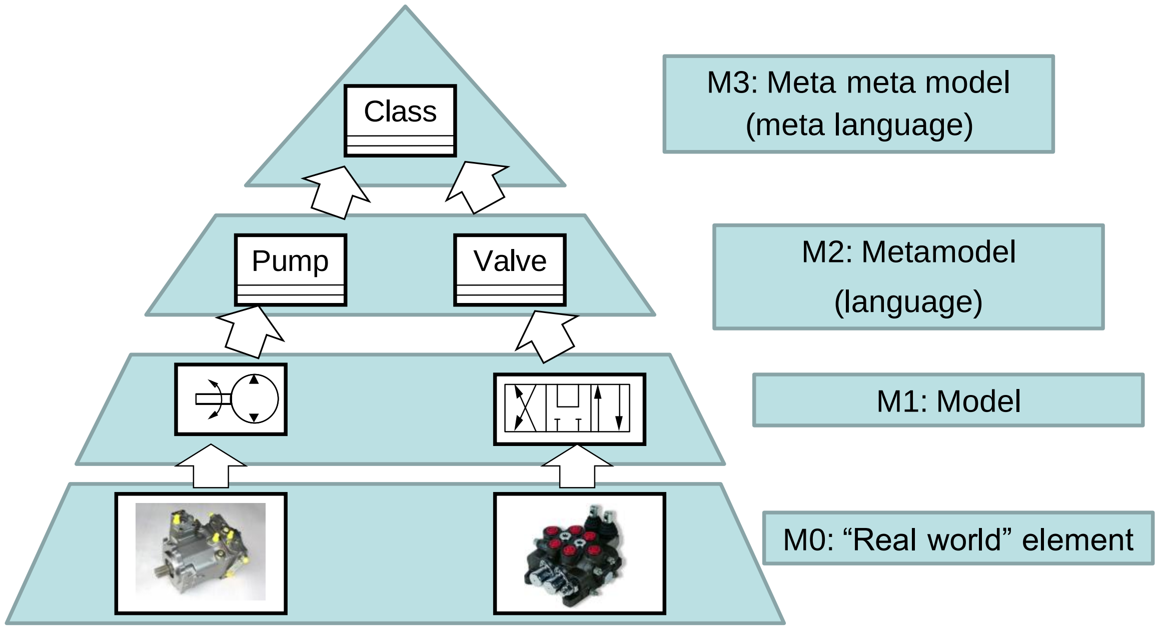


Figure 2.1 : Relationship between metamodels, metamodels and metalanguages

2.1.3 Domain-Specific Languages

A *domain-specific language (DSL)* is a language designed to describe a particular problem domain. It serves the purpose of making the key aspects of a domain – although not all conceivable content – formally expressible and modelable. [62]. A DSL possesses a metamodel as well as a corresponding concrete syntax. The semantics of the DSL are also required to give meaning to the constructs of the metamodel. The modeler must know the meaning of the language elements in the DSL to be able to create reasonable models. Also, model transformations must be able to exactly execute the semantics of the DSL. The semantics of a DSL must be either well-documented or intuitively clear to the modeler. This is made easier when the DSL adopts concepts from the problem space so that a domain expert can easily recognize it [17, 62]. In MDSD, these domains often deal with specific software architecture.

2.1.4 Transformations

Model transformations in MDSD are always based on a metamodel. It is common to distinguish between model-to-model transformations where the transformation creates a new model typically based on a new metamodel and model-to-platform or model-to-code transformations where code is generated that fits into the existing framework. [62]

2.2 Performing Model Transformations

Model transformations, as conceptualized in the graph depicted in Figure 2.2, are anticipated to play a major role in future MBSE endeavors [62].

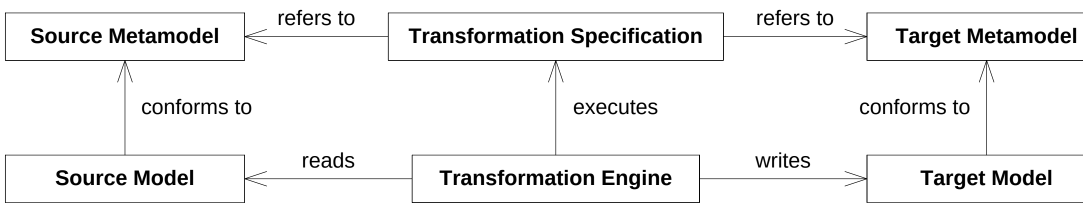


Figure 2.2: The basics concept of model transformation [62].

Generally, model transformations are performed by transformation engines that can read a source model conforming to a source metamodel and execute a transformation specification to produce a target model conforming to a target metamodel. Current applications of model transformations include model synchronization and the generation of low-level models/code from high-level models.

Many methods exist for completing model transformations between two or more modeling languages (metamodels). Two common transformation tools are OMG's Queries/Views/Transformations (QVT) [43] and Triple Graph Grammars (TGGs) [58].

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 source and target metamodels at different levels of fidelity. The *Operational Mappings* language is then used to perform imperative transformations based on the relationships depicted in the *Core* or *Relations* languages. The relations between the QVT languages are depicted in Figure 2.3.

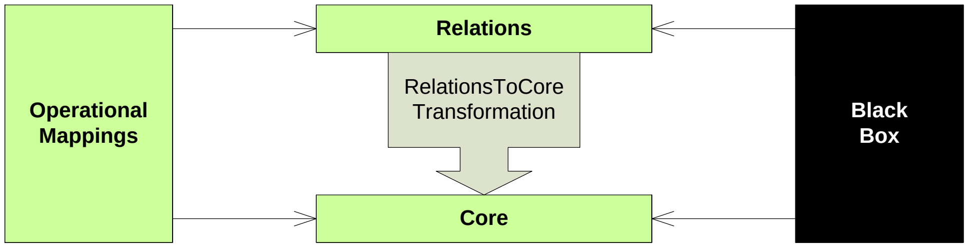


Figure 2.3: Relations between the QVT languages [43].

Overall, QVT is a powerful and widely accepted model transformation tool; however, the imperative nature of the *Operational Mappings* language hampers bidirectional transformations.

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 [19]. 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 [29].

2.3 Standard Ways to Define DSLs using UML

DSLs are a major part of the work presented in this thesis; several methods to formally define DSLs are presented in this section. There are several standard ways that DSLs are defined in MDSD and the software development process. [71]. OMG has introduced *profiles* as a light-weight mechanism to extend UML. Also, OMG provides the Meta Object Facility (MOF) [43] 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 is ambiguities concerning inheritance between stereotypes and also validation of constraints does not work properly in general[71].

UML can also be extended through the use of a MOF tool and the merge concept from the UML Infrastructure [23]. 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 [71]. This approach is illustrated in Figure 2.4.

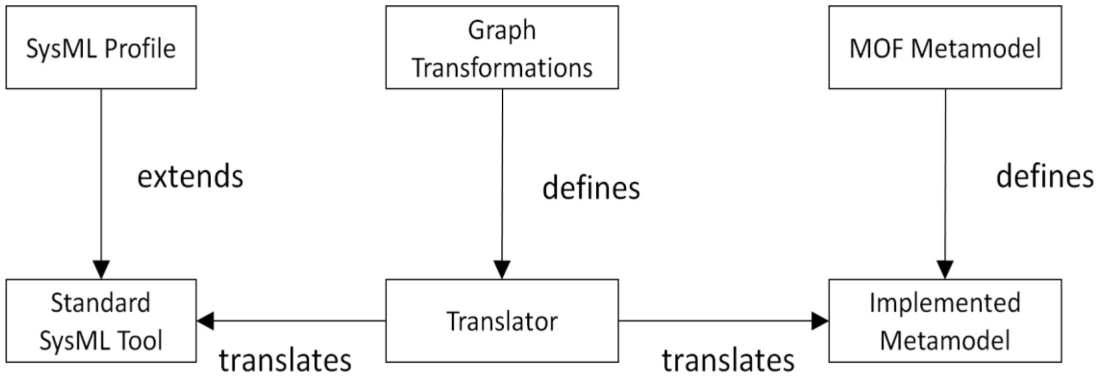


Figure 2.4: A combination of UML profiles and metamodel based technologies

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 QVT based on TGGs is used to translate the stereotyped UML model into an instance of the metamodel.

This approach has the benefit of being both expressive and quickly implementable to provide tool support. In this thesis, this approach is extended with the use of SysML instead of UML.

2.4 An introduction to SysML

SysML is used extensively in this thesis as a foundation for the concrete syntax of DSLs. SysML is an extension of the Unified Modeling Language (UML) [23]. UML is standardized by the OMG and which is currently commonly used in software engineering practice. This section provides a brief introduction to some of the entities from SysML used throughout this thesis.

2.4.1 SysML Blocks

The primary modeling unit in SysML is the *block*. A block is a modular unit of a system description. [42], a block is a modular unit of a system description. A block can represent anything, whether tangible or intangible, that describes a system. For instance, a block could model a system, process, function, or context. When combined together, blocks define a collection of features that describe a system or other object of interest. Hence, blocks provide a means for an engineer to represent a system by decomposing it into a collection of interrelated objects.

2.4.2 SysML Flow Ports

A block's interfaces are commonly defined through the use of *flow ports*. A flow port specifies the input and output items that may flow between a block and its environment. [42] Flow ports are interaction points through which data, material, or energy can enter or leave the owning block. The specification of what can flow is achieved by typing the flow port with a specification of things that flow.

2.4.3 UML Profile and Stereotypes

A *stereotype* is a UML construct used to create customized classifications of modeling elements. Stereotypes are commonly organized within *profiles*. Profiles are a feature that SysML shares with UML; they allow users to specify constructs that are less abstract and more precise by specializing existing SysML entities. Stereotypes are defined by keywords that appear inside of guillemets (e.g., “<<Block>>”).

CHAPTER 3

CAPTURING SYNTHESIS KNOWLEDGE

This section presents a framework for the systematic encoding of synthesis knowledge and the application of this synthesis knowledge to generate design alternatives in an effort to answer Question 1 presented in the introduction. The use of formal models provides an unambiguous and common protocol for communicating design information among various stakeholders. It also facilitates the storage of design information in a form that is computer interpretable making it possible to leverage related work in computer science. It also promotes traceability throughout the design process by employing models as a form of documentation.

This framework relies on the definition of several domain specific languages (DSLs) through the use of metamodels and model libraries to capture the synthesis knowledge about a particular domain. A DSL is a language that is tailored to describe a particular problem domain. In this approach, the language is augmented by the specification of graph-based model transformations designed to *transform* models of a systems engineering problem into models of a specific design alternative. Specific design alternatives are automatically generated by applying these graph-based model transformations to models also defined by the same DSL. 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. The use of graph-based transformations also has the advantage of being easily visualized.

Various methods are presented in the literature for using design grammars to provide automated synthesis to explore the design space of a particular problem [6, 7, 11, 56]. Although graph-based synthesis methods have been shown to be capable of finding an optimal or near-optimal design solution [56] within a given design space, how to specify this design space is taken largely for granted or defined in an ad hoc manner. The representation of structures used in design generation and evolution using formal languages and graph-grammar concepts has been explored [1, 63], although through the use of global formalisms and languages. Global languages have the disadvantages of being less precise and more ambiguous because they need to have the flexibility of defining structures in a nearly infinite number of possible domains.

Instead, the approach of defining languages that are domain specific is taken. These languages can be more precise because they only need to capture a small number of coupled domains. The disadvantage of a DSL is the additional effort required to define and implement the language. To mitigate this disadvantage, the thesis explores implementing these DSLs using concepts applied from Model-Driven Software Development (MDSD) which allow for the automated generation of computer code [62], reducing the expense.

Also, many of the previously mentioned approaches require problem-specific computer code for the generation and execution of analysis models. Instead, by capturing possible design alternatives in formal models, the creation of corresponding analysis models can be automated [27].

The formal capture of synthesis knowledge using a model-based approach does have disadvantages: there is a higher level of expertise and effort required to explicitly capture knowledge that would otherwise be assumed implicitly. In this thesis, it is explored how this disadvantage can be offset by employing concepts of *modularity* and *composition*. A modular modeling approach is taken to describe synthesis knowledge as a set of the possible modular components that may appear within a system and the possible connections between those components. Port-based models [45] are used to describe the possible components; these models are then integrated into more complex systems by creating connections between well defined interfaces. This fits naturally with current systems engineering practice which relies on composition and integration to manage complexity by decomposing complex systems into modular chunks that can be easily reused and reconfigured.

3.1 Representing Design Alternatives using a DSL

In this chapter, the approach is presented to capturing synthesis knowledge through the use of formal models and how design alternatives can be generated from this knowledge. Specifically, the synthesis knowledge that is captured describes how to define a design space and generate possible design alternatives. The design spaces of interest stem from a large number of systems engineering problems involving the composition of well-defined components into more complex systems. This definition is derived from the view of common systems engineering problems. When the design space is described in this manner there are several pieces of knowledge that naturally appear necessary to formally capture:

- What are these well-defined components? What are their functions and interfaces?
- How can these components be connected together? How does the designer combine these components to generate meaningful design alternatives?

A formal DSL must be defined before these aspects can be formally captured in models. There are several standards-compliant ways to define DSLs [71] but, in general, an abstract and concrete syntax need to be defined. The next section describes the definition of the abstract syntax through a metamodel followed by the definition of the concrete syntax by extending SysML with a profile.

3.1.1 Defining the Abstract Syntax

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 [15].

Metamodels are defined through the use of a metamodeling language; this metamodeling language is in turn defined by a meta-meta-model. Although this meta-hierarchy could continue ad infinitum, practically speaking metamodeling languages describe themselves through meta-circularity [17]. The metamodeling language used in this thesis is OMG's Meta-Object Facility (MOF), a language designed by OMG for Model-Driven Architecture (MDA) [43].

Metamodels need to be formal and unambiguous by having a unique and precise meaning that is defined by a mapping from the metamodel into a semantic domain. This semantic domain is the design space of a particular systems engineering problem. A generic metamodel can be specified; because a systems engineering based approach is taken, this space is spanned by systems, each system containing at least one component. The approach to modeling components is port-based; this is reflected by defining that each component can have any number of ports. These ports can be connected to each other; sometimes more specifically one port can be connected to exactly one other port. Generally each component is part of only one system and each port is only owned by one component. The metamodel analogous to the description just provided is shown in Figure 3.1.

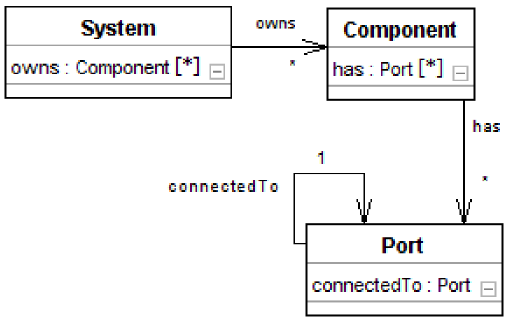


Figure 3.1: A visualization of the generic Metamodel defined using MOF.

Once this generic metamodel is defined it is extended to more precisely capture the domain of interest. The types of systems, components, ports, and connectors that appear in the design space are defined using specialization relationships. For example, a specific type of component is a specialization of the generic component. If the systems engineering problem was the design of a mass-spring-damper (MSD) system then the metamodel is extended to include constructs for defining a spring, a mass, and a damper. This is shown in Figure 3.2. The specialization relationships are illustrated as a solid line with a hollow arrowhead. In this example, the spring is a specialization of the generic class of components. These new constructs can be more precisely defined by further specifying the types of relationships that may exist between them.

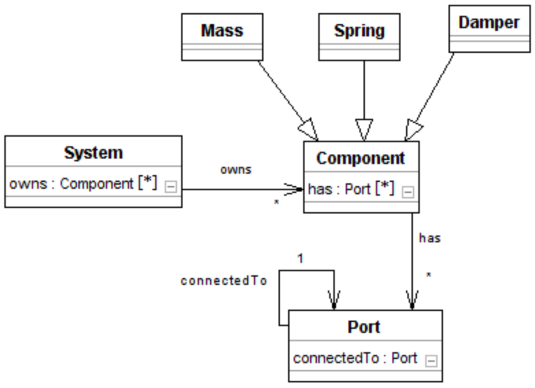


Figure 3.2: Metamodel extended to capture the types of components that can exist in the MSD system

The metamodel can be validated by using its terminology in all discussions with domain experts and stakeholders [13]. It can be considered as a grammar for building valid sentences in the respective domain. Several sentences fall naturally from the definition of the metamodel in Figure 3.2:

- A component has any number of ports.
- There can be three types of components in this system: masses, springs, and dampers.
- Each port can be connected to exactly one other port. (This final consideration may not be applicable for all systems. In the example, an entity is created for connectors that facilitate the connection of any number of ports to each other.)

If a valid design alternative in the design space cannot be precisely expressed using this metamodel then the metamodel is imprecise and needs to be extended. The abstract syntax only defines the “essence” of the domain specific language: the available constructs and their relationships. To use the DSL to define models a concrete syntax is also needed.

3.1.2 Implementing the Concrete Syntax

After the metamodel is defined, the DSL is implemented by defining the *concrete syntax*. The concrete syntax is the textual or graphical constructs with which the modeling is done. SysML is used to provide the foundation for the concrete syntax. It was developed by OMG to support MBSE; it is a general-purpose systems modeling language that enables the creation and management of models of engineered systems using well-defined visual constructs.

The constructs provided by SysML are extended through the use of a profile. SysML is an extension of the Unified Modeling Language (UML) [23], which has been standardized by the OMG and which is currently commonly used in software engineering practice. Profiles are a feature that SysML shares with UML; they allow users to specify constructs that are less abstract and more precise by specializing existing SysML entities. The profile is defined by extending the *block* construct of SysML. The block is the primary modeling construct of SysML; it can represent anything, whether tangible or intangible, that describes a system.

There are several further steps taken to implement a DSL derived from [71] by adapting an existing SysML modeling tool:

1. The abstract syntax of the domain specific language is captured in a MOF-compliant metamodeling tool as described in the previous section. In this case, the chose is MOFLON [2] as the meta-modeling tool because of its code generation capabilities.
2. A SysML profile is defined within the SysML modeling tool. This profile has a one to one mapping to the specified metamodel and can be used to stereotype a particular SysML model. For example, the profile of the metamodel for the MSD system described in the previous section is shown in Figure 3.3.
3. MOFLON is used to generate Java Metadata Interface (JMI) based code that implements the metamodel.
4. Query/View/Transformation (QVT) based transformation rules are also defined in MOFLON to map between the stereotyped SysML profile and a specific instance of the metamodel. This serves the role of a translator or compiler between the concrete syntax and the abstract syntax.
5. MOFLON is used to generate JMI code that implements these transformations.
6. The code generated by MOFLON is combined with a JMI-compliant SysML tool. This extends the tool to provide the capability of authoring models defined by the DSL.

A SysML model is stereotyped using the profile. It is then translated into an abstract representation by executing the JMI code. This abstract representation is an abstract syntax graph; this graph is the abstract representation of the defined model. Graph transformations are then applied to this abstract syntax to generate design alternatives.

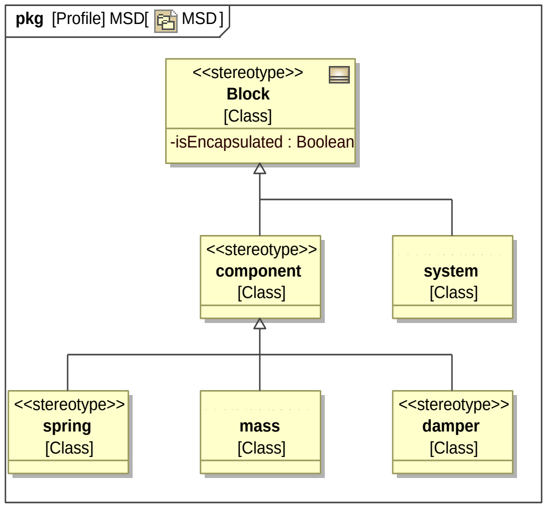


Figure 3.3: Profile used to label SysML entities that corresponds to the abstract syntax defined in the metamodel

3.2 Generating design alternatives

3.2.1 Defining the Graph Grammar

The metamodel defines the space of design alternatives; this section addresses how to create possible instances in this space. As previously mentioned, the goal is to provide domain experts with a framework to express their knowledge that leads to unambiguous definition of potential design solutions as well as effective application of that knowledge using search algorithms.

The metamodel as presented only captures the *syntax* of the domain specific language: an unambiguous way to define potential solutions. It is extended to further capture how design alternatives can be generated. This is accomplished effectively through the use of a graph grammar which provides a structured representation of knowledge using rule-based techniques [40, 53]. A graph grammar consists of a set of graph transformations; in this case, the graph transformations applied in sequence generate a possible model conforming to the metamodel.

These graph transformations have a *left-hand side*, the pattern of a graph that is matched and a *right-hand side*, the replacement graph. They are also defined using the abstract syntax provided by the previously discussed metamodel along with the QVT transformation standard.

The transformations are applied in certain sequences result in models that conform to the metamodel. In order to generate all of these possible models, one must traverse every possible sequence of transformation rules. These transformations are generative; they involve the incremental specification of a design alternative.

The transformations model the addition of components to the system along with the valid ways that components are connected. Although the transformations presented in this thesis do not take into account sizing, similar attribute grammars can be used to capture how components are sized and configured [53].

The transformations match a portion of the model/graph and create new instances of the component types. Although these transformations are defined at the metamodel level, they are executed on instances of the model. Also, use of a domain specific metamodel which involve constructs that designers should be familiar with reduces ambiguity.

Previous work has illustrated the advantage of using visual graph transformations as a guide to generating code [57]. Because these transformations are modeled formally using the QVT standard, MDSD concepts are used to automate the generation of code to execute these transformations on models described by the JMI implementations described in the previous section.

Returning to the MSD system example, there are several transformations that specify a valid instance of the system. One possible transformation instantiates a model of a spring and connects it in parallel with another spring in the system. This transformation is shown in Figure 3.4. It takes in the system model as an input. The left-hand side of this transformation is the fragment of a model consisting of a spring with two ports which are unconnected. The right-hand side is the addition of a second spring to the model in parallel with the first.

A number of such transformations exist for the MSD system: the instantiation of dampers in series and parallel, springs in parallel, and the instantiation of masses. By applying these transformations in one of many possible random sequences, an instance of the MSD system metamodel is generated.

The specification of these transformations at random is sufficient to generate instances of the metamodel, but simply executing the transformations at random is often inefficient in generating alternatives. To further model the order in which these transformations are applied, the "good" orders are modeled through the use of a decision graph. This decision graph is a very simple model of the process a designer goes through to define a design alternative.

A decision graph is an extension of the hierarchical decision tree presented in [39]; unlike a tree, a graph can contain loops. Decisions that a designer makes when creating a design alternative are modeled as nodes; they are connected by edges that describe the order in which the decisions are made. Each node is mapped to the representative model transformation.

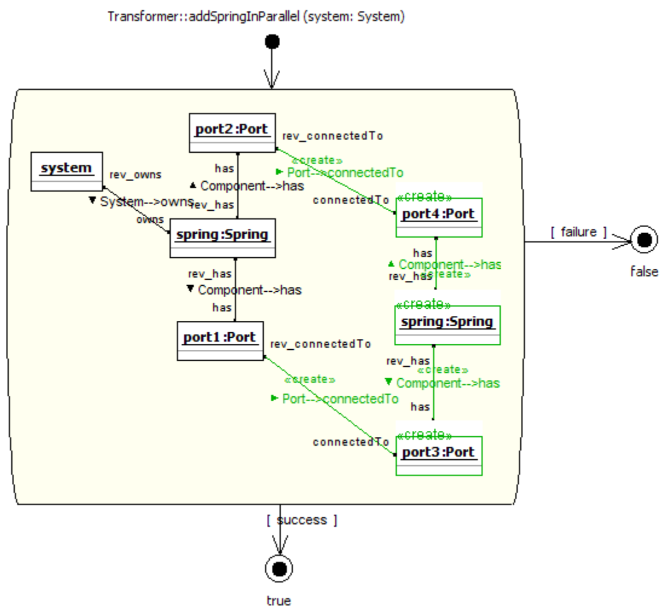


Figure 3.4: Graph transformation that adds a spring in parallel with another spring to the system.

By traversing the decision graph and executing the corresponding transformations, a complete model of a design alternative is created. This alternative is also represented using the abstract syntax and can be translated into a concrete representation or a corresponding analysis model.

3.2.2 Capturing Fragments in a Model Library

The language is further defined by enumerating exactly which *instance*s appear by capturing them within a model library. A model library contains useful fragments which can be composed into more complex models. The metamodel is only a definition of the *types* of constructs and relationships that appear in a DSL: the types of physical structures that appear within the domain specific design space. This model library is the vocabulary of the DSL: the models of physical structures that can be combined to create valid design alternatives.

To fit within common and current systems design practice, the majority of models appearing in the model library are the modular components (or subsystems) that need to be integrated. These models are port-based; they clearly capture the interfaces that can be used to connect one model to another. Also each model that appears within the model library should have a corresponding type definition in the metamodel. The models are created within a SysML authoring tool using the concrete syntax previously defined.

Along with the interfaces, compatibility between components is also explicitly captured. Models of components that are compatible are organized into sets. This addresses the case when compatibility cannot be determined simply by examining the interfaces of a component. A fairly strong assumption is made when grouping components into these sets: any component within the set is compatible with all of other components within the set. For the examples presented in this thesis this assumption holds, but further investigation is needed to test if this approach is truly sufficient.

The model library needs to be validated through discussion with stakeholders and domain experts just as the metamodel is validated. The library is organized into a component taxonomy to facilitate exploration by designers and stakeholders.

The knowledge contained in appropriate models from this library is transferred to the new model instances created during the generative model transformations. For example, when a model of the spring is instantiated knowledge from a spring model in the library is also associated with it. If there are multiple appropriate models (multiple instances of the same component type), one can be selected at random from the library or the metamodel can be extended to characterize each instance with a separate component type.

3.2.3 Searching the Design Space

In this section, a method is presented to search the design space defined by the DSL through the use of an evolutionary program [32, 35]. One goal of capturing synthesis knowledge is automating the design process by applying a search algorithm to the design space. An evolutionary program is a global stochastic optimization technique that has the advantage of being largely problem independent. It can be used on design spaces without well-defined distance metrics, although a fitness function is needed to compare solutions. They are similar to genetic algorithms [18, 21] but involve the use of problem-specific data structures. In this case, these data structures are models defined by the DSL.

Evolutionary programs are designed to mimic the evolutionary process: a population of solutions is iteratively modified over multiple generations with the goal of increasing the population fitness and the quality of individual solutions. Evolutionary programs maintain a population of possible solutions; an initial population is generated at random. Naturally, for design synthesis the population consists of design alternatives. An initial population is created by synthesizing several design alternatives from the captured knowledge. Each possible solution is evaluated using a fitness function. The next population is created by modifying selected possible solutions from the previous population. There are several selection techniques, but a fitness proportionate selection

[32] scheme is used here where solutions with higher fitness have a higher probability of being selected. These solutions are modified using either crossover or mutation operations: in a crossover operation characteristics from two possible solutions merged into single solution; in a mutation operation one solution is modified into a new solution.

So far, the captured synthesis knowledge has been problem independent. To generate design alternatives that are specific to a given problem, an embryonic model [31] is used. This model is an incomplete instance of the DSL, it is fragment of a potential design alternative which is required for the design alternative to be applicable to the specific problem. Model transformations are applied to this embryonic model until a design alternative is fully specified. The order of these model transformations is determined by the selection of a path of nodes from the decision graph; this is analogous to the process a designer would use to create an alternative. Every model transformation adds instances from the model library; these instances match the types defined in the transformations but are chosen at random. To uniquely define a single alternative, the path taken through the design graph and the random instances added by the transformations are required.

One convenient aspect of using a sequence of transformations is that one is able to serialize the graph representation by simply capturing this sequence of transformations applied. This allows the modification of possible design alternatives without needing to specify additional model transformations. Using a standard evolutionary approach, standard mutation and crossover operators are applied to these serialized representations to modify the design alternative and search the space.

To implement the optimization algorithm, an efficient way to represent each design alternative is needed. Since each design alternative is represented as a sequence of transformations, the design alternatives are represented as a set of numbers that reflects the sequence. For each node in the classic mutation and crossover operators are applied to this set of numbers resulting in modifications to the design alternatives. For the mutation operator, instead of simply fitting a bit, a new random number of the sequence is generated. Although this is inefficient, this allows the maintenance of the probabilities in the design graph. After the set of numbers is modified, the transformations are applied in the new sequence. To insure the specified design alternative is completely specified, the sequence of numbers must result in a sequence of transformations that terminates at the end node of the decision graph. If the set of numbers specified a sequence that terminates prematurely, additional transformations are applied until the end node is reached. The sequence of these transformations is added to the set of numbers describing the design alternative. This method is applied on the hydraulic circuit example in Section 3.4.

3.3 Example: Hydraulic Circuit Generation

The synthesis approach is applied to the design of a generic hydraulic circuit. From a systems engineering perspective, hydraulic systems have the interesting characteristic that they are circuit-like; that is, they consist of discrete components that are configured or composed into complex systems. This modularity in the physical system has been introduced to facilitate their design and manufacture. Modular components not only provide economies of scale in the manufacturing process, but they also simplify the design problem by decomposing the system into functional units that have simple and clearly defined interfaces through which they interact with each other.

The hydraulic circuit provided in this example is similar to circuits common in off-road equipment. The requirement placed on possible circuits is that they must actuate exactly four loads. These loads are an abstraction of the mechanical structure of a possible piece of equipment. An assumption made by the problem formulation is that the directional valves being modeled have common valves bundled with them. Several additional constraints are assumed from the problem formulation:

- Every port must be connected to at least one other port.
- There is at most one pump connected to one valve.
- Every actuator is connected to exactly one valve.
- Each load has exactly one actuator.
- Each directional valve must receive hydraulic flow from a suitable pump. (Namely, closed-centered load sensing valves must be connected to a variable displacement pump and open-centered valves must be connected to fixed displacement pump.)

3.3.1 Domain Specific Language

To start, a DSL for capturing possible hydraulic circuit topologies is created. The abstract syntax of this language is specified in a MOF-compliant manner with the MOFLON tool. This new metamodel is an extension of the generic meta-model presented earlier in Figure 3.1. Several entities are created to capture different component-types that are commonly found in a hydraulic system: entities for labeling pumps, cylinders, directional valves, tanks, relief valves, and boundary components. Boundary components include anything initially specified in the embryonic model; the “boundary” of the circuit. That includes the number of loads that must be actuated and the number of power sources that can be used to drive the pumps. This metamodel is shown in Figure 3.5; it is not inclusive but can be extended to apply to more complex problems. The abstract syntax is implemented through the automatic generation of code from MOFLON. A concrete syntax is defined in a SysML tool using a profile. The code to translate between the concrete syntax and an abstract representation is also automatically generated.

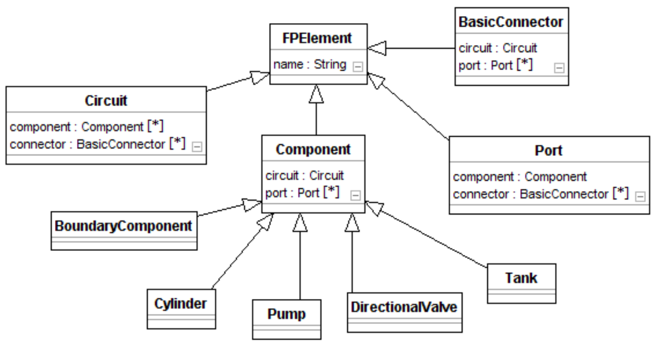


Figure 3.5: Visualization of hydraulic circuit metamodel used to define abstract syntax of domain specific models.

3.3.2 Graph grammar

Graph transformations are defined to capture common connectivity between component-types; the set of these transformations make up the grammar. These transformations are also defined within MOFLON using the abstract syntax. These transformations reflect actions designers might take to create a hydraulic system. Graph transformations are defined to:

- Add an instance of a cylinder to the circuit and configure it to actuate a load.
- Add an instance of a directional valve to the circuit to control a cylinder.

- Add an instance of a pump to the circuit to provide flow to the directional valve.
- Add an instance of a tank to the circuit to provide flow to instances of pumps.

The overall structure of these transformations is similar. For example, the transformation defined to add a cylinder to the circuit is shown in Figure 3.6. The complete set of transformations are included in Appendix B. The left-hand side of the transformation is a boundary component that owns an unconnected port of the appropriate type. The right-hand side of the transformation is the new actuator and connectors. The transformations are designed to maintain the first three constraints specified by the problem formulation.

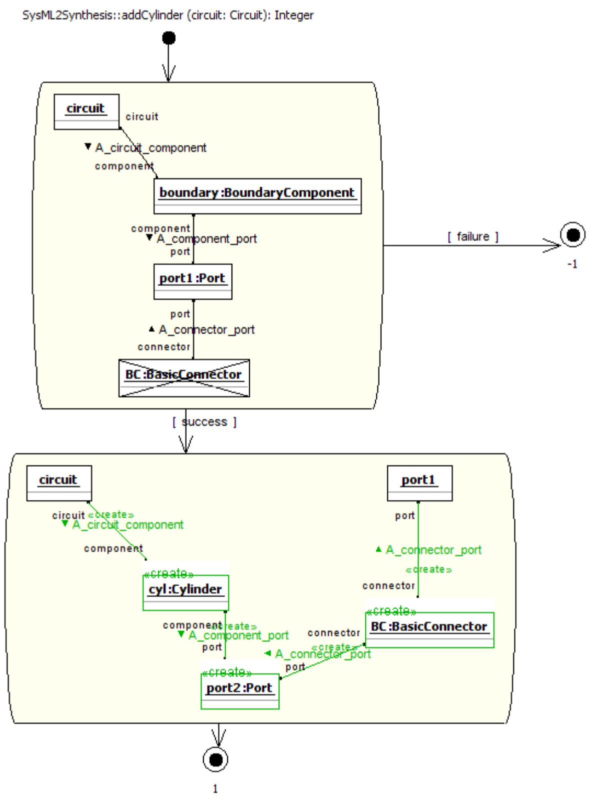


Figure 3.6: Graph transformation rule to add a cylinder model to the system

model and connect it to an appropriate load.

3.3.3 Model library

In this example, the enumerated number of possible components remains small. This is because of the chosen abstraction level: each component has an implied structure but not sizing parameters. The use of port-based models meshes with the modeling of hydraulic components because it reflects the true nature of the system. The models are broken into two compatibility groups: one for components that can be connected to closed-center valves and one for components that can be connected to open-center valves. Modeling these compatibility groups allows the last constraint of the problem formulation to be met.

3.3.4 Encoding problem specific knowledge

The problem specific knowledge is encoded in an embryonic model. In this example, the problem specific knowledge is the number of loads to be actuated. If the circuit cannot actuate these loads, it is invalid and is not considered because considering impossible solutions is inefficient. Therefore, the embryonic circuit contains four loads. Random instances are generated when the set of graph transformations are applied to this embryonic model and all of these instances will actuate exactly four loads.

3.3.5 Decision graph

The possible sequences of transformations are represented in a decision graph. This graph is shown in Figure 3.7. Each node of this graph corresponds to a previously defined transformation. Probabilities are assigned to the edges of the graph to increase the chances of certain transformation sequences.

The overall layout of the graph is based on one possible sequence of decisions a designer might make to design a single circuit. The graph is represented using a formalism similar to flow charts. Each edge has a probability associated with it; by adjusting these probabilities the likelihood is changed that specific sets of graph transformations are used to generate a design alternative.

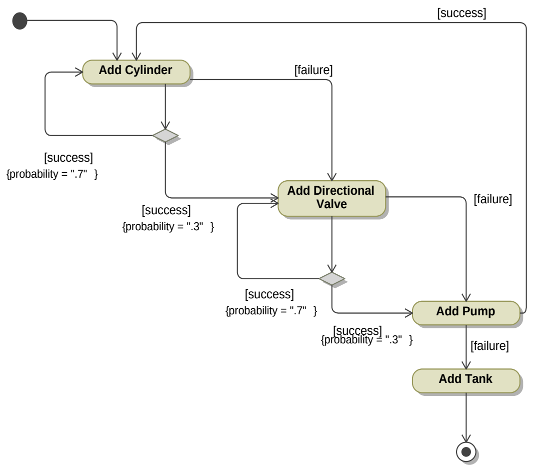


Figure 3.7: Decision Graph for the hydraulic circuit example.

Each node is tied to a graph transformation and each edge has probabilities associated with it.

3.4 Results

There are several considerations when exploring the effectiveness of generating alternatives from the captured synthesis knowledge: Are the generated design alternatives valid? Do these alternatives span the space uniformly? And, how well does a search algorithm perform when searching through the space?

The approach is first tested by generating a basic hydraulic circuit: one that needs to actuate only a single load. The initial embryonic model includes a single load and single power source. The graph transformations described earlier are applied to the circuit. The result is shown in Figure 3.8. The circuit is one with a single variable displacement pump connected to a closed-center directional valve. As mentioned previously, the directional valves modeled include common valves such as relief valves from high pressure to low pressure flow. The circuit is valid: all the hydraulic ports are connected, the variable displacement pump and closed-center valve are compatible, and one cylinder to actuate the load. A large number of more complex circuits have also been generated; all the generated circuits satisfy the prescribed constraints.

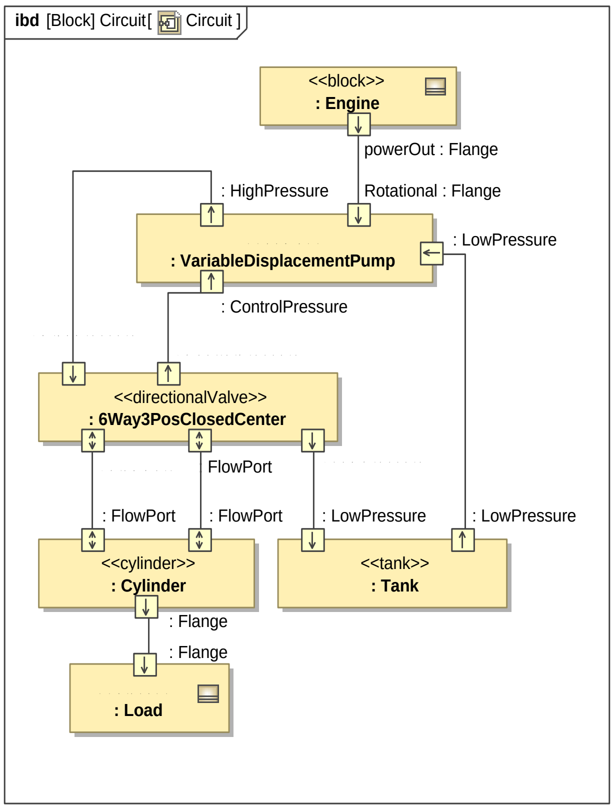


Figure 3.8: Simple hydraulic circuit represented using concrete syntax in

The next consideration is whether the alternatives span the design space. To test this in the example problem, a number of random design alternatives are generated and characterized based on their topologies. The requirement of the circuit actuating four loads along with the assumed constraints implies that each circuit should have between one and four pumps. The number of pumps is also a characteristic of the circuits that is unambiguous and easy to measure. The number of pumps per alternative for 1200 random design alternatives is shown in Figure 3.9. The first 600 design alternatives are generated using the decision graph in Figure 3.7 where the probabilities are labeled along the edges. The probabilities in the decision graph bias the generation process towards the generation of alternatives with fewer pumps because it is more common to find fewer pumps in real world systems, specifically the edges leaving the “Add Directional Valve” node. When evaluating these edges there is a probability of .70 that an additional directional valve will be added to the circuit and connected in series with other valves if possible whereas there is only a probability of .30 that an additional pump will be added. The next 600 design alternatives are generating using a decision graph further biased to generate alternatives with fewer pumps by adjusting the previously mentioned probabilities from .70 and .30 to .90 and .10 respectively. These 600 alternatives on average have fewer pumps than the first 600.There are simply more possible configurations with two pumps than the fairly limited number of one pump configurations.

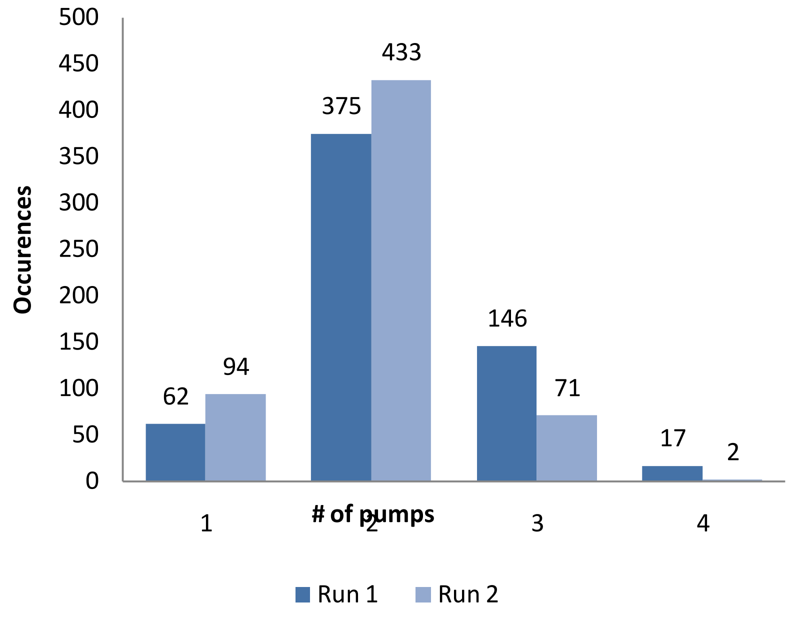


Figure 3.9: Results from random synthesis of alternatives

To test the performance of the evolutionary search algorithm described in Section 3.2.3, it is used to find certain prescribed topologies. In particular, the topology of interest is the rarest topology generated during the previous experiment: a circuit with four closed-centered valves each connected to exactly one pump. The fitness function used to evaluate the design alternatives based on the number of pumps and closed-centered valves is shown in Equation 1:

$$\text{fitness} = (\text{\# of pumps}) + 3 \times (\text{\# of closed-centered valves}) \quad (1)$$

An arbitrary weight is placed on the number of closed-centered valves. This is a crude approximation of the preferences, i.e. that a circuit with 4 closed-centered valves and 3 pumps is closer to the true solution than one with 4 pumps but only 3 closed-centered valves. Clearly, for the four actuator case the maximum possible fitness is 16. Also, each population consists of exactly ten circuits.

In general, the overall fitness of the population improves as the algorithm progresses. Because the evolutionary program is a stochastic process every run does not return the same result. The average progress of 100 runs of the evolutionary algorithm is shown in Figure 3.10. The median maximum fitness and average fitness are shown along with 25th and 75th bounds for the maximum fitness. In general, all the runs converge to the maximum possible fitness, usually over a relatively small number of generations. One aspect of future work is to characterize the performance of the search algorithm if a behavioral-based fitness function is used instead of a topology-based function. In order to accomplish this, simulation models need to be created from structural representations to analyze the behavior of the topologies.

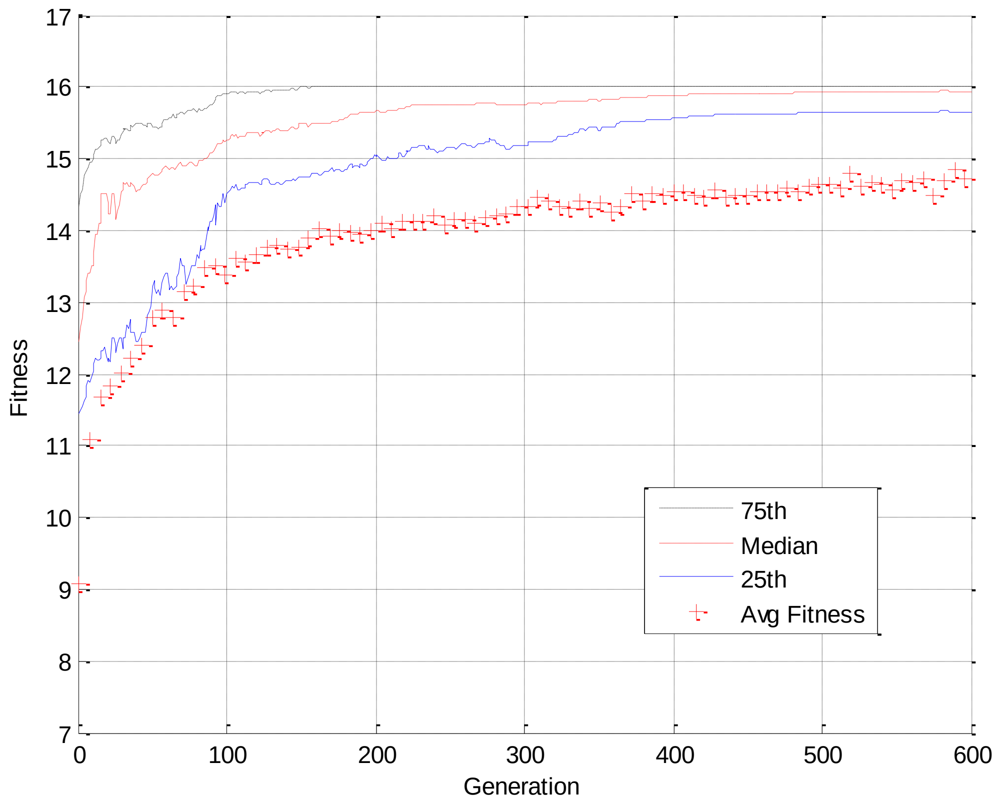


Figure 3.10: Average progress of evolutionary program over 100 runs with average fitness and median maximum fitness for each generation

3.5 Discussion

The approach to generating design alternatives is based on the definition and application of a set of graph transformations (the graph grammar). These graph transformations are a part of the DSL's metamodel. This approach is taken because design alternatives can be efficiently generated through the application of these graph transformations. Many of the constraints placed on the design alternatives are therefore implicitly encoded in these transformations.

An alternative approach is to define the metamodel with these constraints explicitly captured through the use of a constraint language (e.g.; the Object Constraint Language (OCL) [59] that is used to describe constraints that apply to UML). This approach may be advantageous because the metamodel may be simpler to formulate. The difficulty becomes finding models that satisfy these constraints and applying search techniques to these models to explore the design space. New modeling tools, such as Alloy [24], can instantiate some or all models that correspond to a metamodel defined by a set of constraints but leave open the question of how search techniques can be applied to these models.

It is often the case that when there are two possible approaches, a hybrid of these two approaches can stress individual advantages while negating disadvantages. Whether this is the case here deserves further investigation, although currently a clear method to combine the two approaches beneficially is not available. Constraints could be encoded both implicitly within graph transformations as well as explicitly in a constraint language, but the advantages of such a hybrid approach needs further exploration.

Also, the level of abstraction of the design alternatives being generated deserves further consideration. In the example, the circuit topologies generated are at a very high level of abstraction. Clearly models can be captured in the library at different levels of abstraction. Also, the circuits generated can be less abstract (containing information about specific off the shelf components, not instances of a generic type of component.) Future work will extend the presented approach to use attribute grammars [53] to size the components. Many systems also require the design of controllers to fit with each topology.

The automated generation of analysis models and simulations for the structural models of the design alternatives also deserves further investigation. Graph transformations have been used to accomplish this sort of model integration.

3.6 Summary

In this section, a method has been presented to define DSLs and generate design alternatives from the knowledge captured within them. The abstract syntax of a DSL was defined using a formal metamodel specified using the MOF metamodeling language. A process was also shown for defining the concrete syntax by extending SysML. It is also shown that graph grammar can be defined using the abstract syntax of the DSL to generate design alternatives. The method was demonstrated on the generation of simple hydraulic circuits.

CHAPTER 4

CAPTURING ANALYSIS KNOWLEDGE USING MASCOMS[1]

This section specifically focuses on the capture 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.

One goal of this section is to shift the cost-benefit balance in favor of formal modeling by reducing the modeling costs. In this chapter, how the use of the concepts of *modularity*, *reuse,* and *composition* can shift the cost-benefit balance in favor of formal modeling by reducing the modeling costs is explored. By reusing the models, certain costs are incurred only once at the time the model is initially formulated and can then be amortized over multiple reuses of the model.

Common systems engineering problems involve the configuration of well-defined components into more complex systems. In particular, this chapter focuses on capturing the analysis knowledge needed to create a system-level analysis model for such a composed system.

---

[1] Based on work by Jonathan Jobe [25]

[2] Based on work by Tommy Johnson [28]

It is interesting to note that while model reuse can enable the cost effective generation of formal systems engineering models, model reuse itself must rely on formal modeling: One can only enable reuse by formally capturing the model, its characteristics, and the contexts in which it can be used.

This section presents a framework for the systematic encoding of analysis knowledge and the application of this analysis knowledge to generate system-level analysis models from system-level structural representations. The use of formal models provides an unambiguous and common protocol for communicating design information among various stakeholders. It also facilitates the storage of design information in a form that is computer interpretable making it possible to leverage related work in computer science. It also promotes traceability throughout the design process by employing models as a form of documentation.

This framework relies on the definition of several domain specific languages (DSLs) through the use of metamodels and model libraries to capture the analysis knowledge about a particular domain. The model libraries are composed of containers called Multi-Aspect Component Models (MAsCoMs) described in Section 4.3.1. A DSL is a language that is tailored to describe a particular problem domain. In the approach, this language is augmented by the specification of graph-based model transformations designed to *transform* models of a system-level structural representation into models of a system-level analysis model. 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. The use of graph-based transformations also has the advantage of being easily visualized.

4.1 Related Work

The reuse of modular design elements has been addressed by many. Baldwin and Clark [5] consider the use of a design structure matrix, task structure matrix, and modular operators to capture modularity in a design. Eppinger *et al.* [12] also consider that systems can be decomposed into modules, but note that some systems are integrative in nature. Integrative systems avoid the overhead of modular interfaces and can therefore achieve higher utilities [68] but are much less likely to have reusable elements. These systems are therefore not considered for the direct application of MAsCoMs. Gershenson *et al.* [16] take the perspective of modularity as it applies to the entire life-cycle of a product design. They claim that all components that are of the same modular form (based on function and interface) will undergo the same life-cycle processes. Using component trees to decompose structure, the level of the component being viewed and its level of abstraction have an effect on the view of the modularity of a process in the life-cycle. This also holds true for the selection of a modular equation model to predict the behavior of a piece of structure in a component tree. Although MAsCoMs are also mapped to component structures and processes (defined by aspects), such models of modules must still be stored for reuse.

The idea of reusing design knowledge by storing the knowledge in a repository has been proposed in the past. The NIST Design Repository [66] 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) [14]. 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 [49], for manufacturing process planning, for functional decomposition [30, 64], or for assembly planning [52]. Similarly, the models developed in this section follow the core relationships defined in the CPM, but refine them with 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.

Both the CPM and this section fit into a broader group of research efforts in which the goal is to define an ontology for design. 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.* [69], Kitamura and Mizoguchi [55], and Horváth *et al.* [22]. 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 [20]. They organize the knowledge about engineering analyses models into an ontology, which includes both meta-data (e.g., author, documentation, etc. — similar to the Dublin Core [51]) 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.* [37].

Jobe [25] expands this past work 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 [47]. 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 topologies can be explored quickly [45].

Relating analysis models to form has been addressed previously in work on Design-Analysis Integration (DAI) [47]. Peak *et al.* relate the parameters of analysis models to parameters of design models in a declarative, reusable fashion using Constraint Objects (COBs) or, more recently, using SysML parametric diagrams [49]. In this section, this approach is adopted but only at the level of individual components (see section on Fine-Grained Design-Analysis Relationships). By establishing the relationships between design and analysis models at the component level, the relationships are maintained even when the components are composed into larger systems, thus further promoting model reuse. To enable composition, additional knowledge is needed both about the model interfaces and about the composition process, as is further explained in Section 5. Wallace *et al.* [70] 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—a benefit of modularity that is also exploited by MAsCoMs.

4.2 Representing analysis models using a DSL

In this section, a formal language is defined to describe the space of analysis models that are of interest. As mentioned previously, the view of systems engineering problems is taken as involving the composition of well-defined components into more complex systems. Since current practice in systems design relies mostly on integration of modular components and subsystems, our system-level analysis models are viewed as models composed of well-defined component models. These component models are connected through well-defined interfaces. When the space of interest of analysis models is described in this fashion, several pieces of knowledge appear necessary to formally capture:

- What are these well-defined analysis models? Which components do they represent? What are their interfaces?
- Which analysis models are meaningful to connect together? And how can they be connected together?

A formal DSL needs to be defined to capture these aspects formally in models. The same approach is taken to defining this DSL as in Section 3.1; an abstract syntax and concrete syntax are defined to describe this language. The rest of this section describes the definition of this abstract syntax through the use of a metamodel.

The initial step to defining this domain specific language is defining 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. It represents the structure of the language independent of any particular representation or encoding.

Metamodels need to be formal and unambiguous by having a unique and precise meaning that is defined by a mapping from the metamodel in a semantic domain. In this case, this semantic domain is the space of system-level composed analysis models. Similar to the generic metamodel specified in Figure 3.1, the space of analysis models is spanned by system-level models composed of component-level analysis models; each such system-level model containing at least one component-level model. The component-level models are assumed to be port-based (as previously described they have well-defined interfaces, or ports); therefore the metamodel is defined to show that each component-level model can contain any number of ports.

This metamodel is formally expressed using OMG's MOF similar to the domain specific metamodel described in Section 3.1.1. A visualization of this metamodel is shown in Figure 4.1. This metamodel can be extended to more precisely capture different classes of analysis models, but for the analysis-models presented in this work the provided metamodel is sufficient.

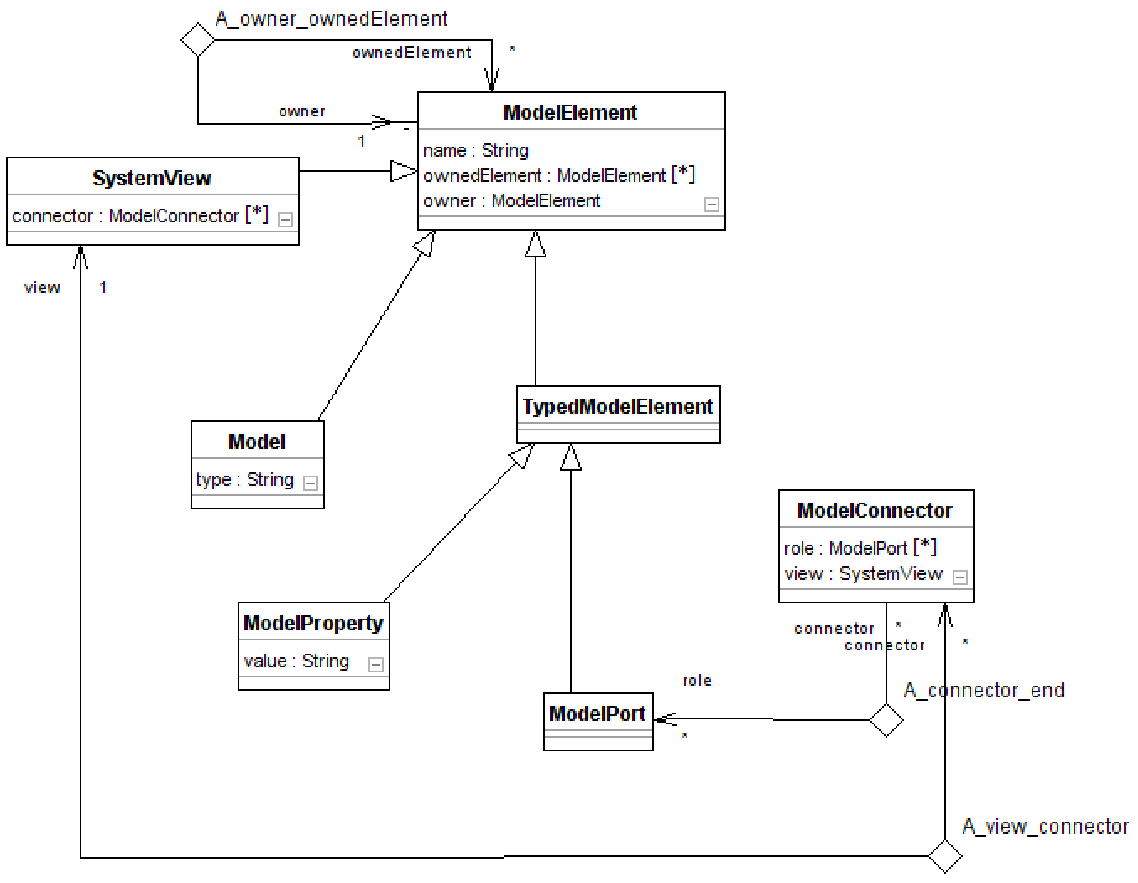


Figure 4.1: Metamodel of DSL for analysis models

Defining a DSL also has the advantage of simplifying the specification of graph transformations because it provides part of the unambiguous language for their expression. Although system-level analysis models as described can be captured using purely SysML concrete and abstract syntax, using a DSL has the advantage of expressing these models in a manner that more concise and often less ambiguous.

The implementation of the concrete syntax is similar to the approach taken in Section 3.1.2 and further described in Appendix. The metamodel and concrete syntax only capture the *types* of constructs and relationships that appear in the space of interest, the next section describes how knowledge about particular instances is captured through the use of a model library containing MAsCoMs.

4.3 Capturing Reusable Analysis Knowledge in a Model Library

4.3.1 Multi-Aspect Component Models

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 an instance level about the well-defined analysis models. The multi-aspect component model (MAsCoM) framework introduced by Jobe [25] 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 models of the structural components they model.
3. How the analysis models model the structural components and which analysis models can be connected together.

The organization of this library takes into account the general view of systems engineering problems previously presented. Analysis models are organized by component type because it follows naturally from the definition of a systems engineering problem 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 is 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 [67] modularity is achieved by implementing cross-cutting concerns separately so that they can be woven into a variety of different software classes. In the context of modeling, rather than weaving models together, what is important is that one can identify which models are compatible with each other so that they can be composed into system-level models. To be compatible, models must 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 defined in a computer interpretable and searchable fashion.

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

4.3.2 A Library of Components

In design, components or subsystem are selected and defined in an iterative fashion. First, a functional architecture is defined after which functions are assigned to components in a physical architecture [54] (or, equivalently working principles and working structures are identified [44]). 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 from a particular company has been identified. At each step along the way, analysis models at different levels of abstraction could be 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.

To support such successive refinement of classes of components down to very specific individual components, it is meaningful to organize the components in a taxonomy. Organizing components into a taxonomy has the additional benefit that one can take advantage of an inheritance mechanism to efficiently associate analysis models with components. For example, in the taxonomy analysis models associated with parents would apply also to children. This raises questions of selecting models of appropriate fidelity and abstraction which are left for Section 6.2.

For the purposes of this thesis, this library of possible components is the same library as in Section 3.2.2. This library organized into a taxonomy is shown in Figure 4.2.

One branch of the taxonomy is illustrated in Figure 4.3 for a pump at various levels at abstraction.

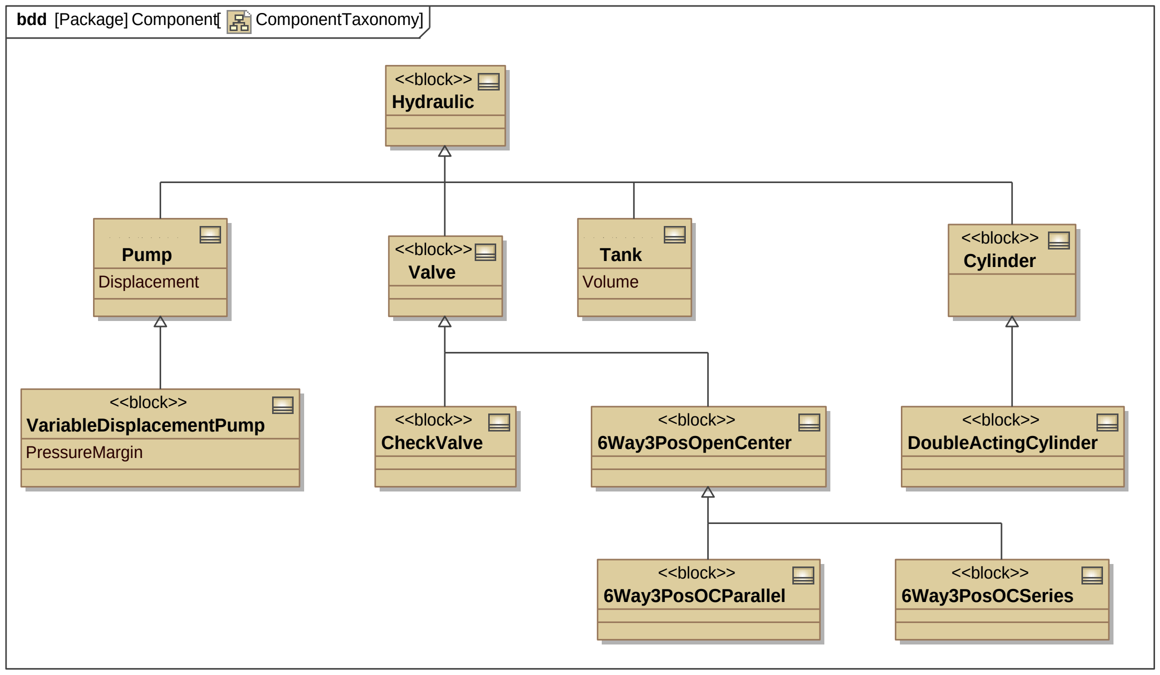


Figure 4.2: A portion of the library of components organized into a taxonomy

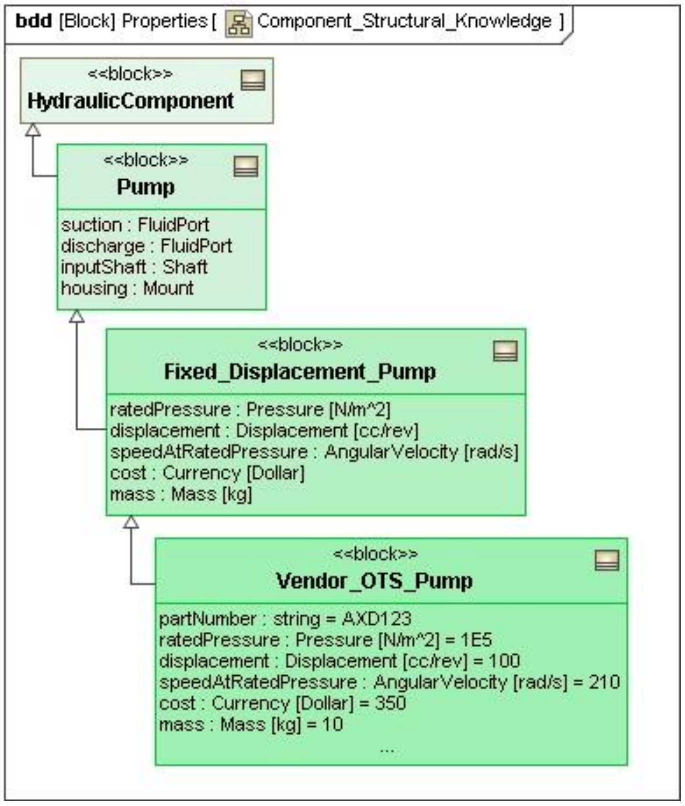

bdd [Block] Properties [Component_Structural_Knowledge]
<<block>>
HydraulicComponent
<<block>>
Pump
suction : FluidPort
discharge : FluidPort
inputShaft : Shaft
housing : Mount
<<block>>
Fixed_Displacement_Pump
ratedPressure : Pressure [N/m^2]
displacement : Displacement [cc/rev]
speedAtRatedPressure : AngularVelocity [rad/s]
cost : Currency [Dollar]
mass : Mass [kg]
<<block>>
Vendor_OTS_Pump
partNumber : string = AXD123
ratedPressure : Pressure [N/m^2] = 1E5
displacement : Displacement [cc/rev] = 100
speedAtRatedPressure : AngularVelocity [rad/s] = 210
cost : Currency [Dollar] = 350
mass : Mass [kg] = 10
...

Figure 4.3: A pump at various levels of abstraction

4.3.3 A Library of Aspects

When reusing a model, one needs to recognize which model is needed from among the many models that may be associated with a particular component. To help the designer do this, models are characterized using aspects. Since there are a large number of potential aspects, it is helpful to organize them also 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 hydraulic and mechanical engineering disciplines, by the continuous time discretization aspect, by the DAE mathematical formalism, and by the Modelica representation syntax. Therefore it should be composed with valve and cylinder models sharing the same aspects.

These aspects formally characterize a model and thus succinctly provide the designer or analyst with 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. Based on the aspects, a designer can be efficiently search or browse through a model repository to identify the model that is most appropriate for a particular design context. 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 not meaningful to combine a discrete event simulation model with a partial differential equation model. Having formal representations of these different aspects available is particularly important when automating the composition process.

4.3.4 Fine-grained Design-Analysis Relationships

The additional knowledge necessary to capture the relationship between the parameters and interfaces of analysis models and the parameters and interfaces of the structural representations in a context-specific instantiation is incorporated in MAsCoMs with two additional constructs: *parameter maps* and *interface maps*.

Parameter maps bind the parameter values in analysis models to the related parameters in the corresponding component's structure 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 analysis models have been associated with components in the component taxonomy, it becomes possible to establish these relationships also in a reusable fashion.

In addition to parameter maps, MAsCoMs also include *interface maps*. Interface maps support the configuration of analysis models for individual components into system-level analysis models. Similar to the composition of structure models into a system schematic, analysis models can be configured into networks through well-defined port-based interfaces [45], as is implemented in tools such as Simulink^{TM} [61], and in languages such as Modelica [38]. Recently, the ability to compose analysis models has even become feasible for finite element models [3, 60]. 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.

4.4 Implementation in SysML

4.4.1 Defining the Language for MAsCoMs

To make the MAsCoMs outlined in Section useful in the context of systems engineering, all the concepts and relationships have been defined in the Systems Modeling Language (OMG SysML^{TM}) [65]. Since SysML has been defined specifically to support systems engineering, it includes modeling constructs that directly support the definition of physical architectures and engineering analyses—the main focus of MAsCoMs.

SysML is the modeling language used to represent MAsCoMs. It is a general purpose language. It provides well-defined visual constructs for modeling system engineering problems. A profile is used to extend SysML to provide additional unambiguous syntax for capturing several unique features of MAsCoMs. This profile is shown in Figure 4.4.

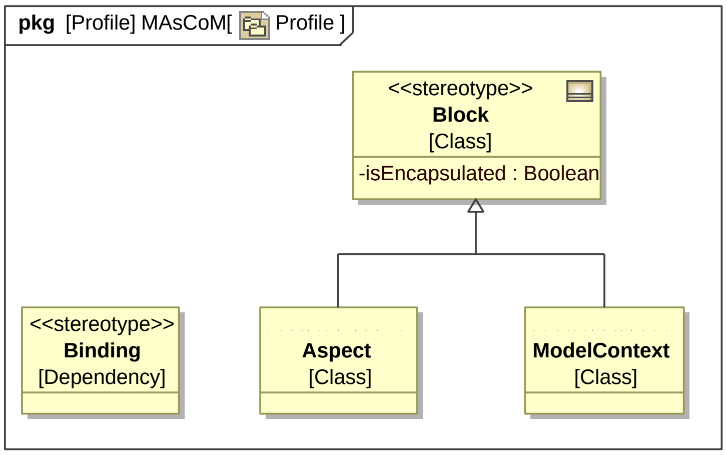


Figure 4.4: MAsCoM Profile

4.4.2 Aspect Library

The aspects are represented using SysML *blocks* that are stereotyped using the <<Aspect>> stereotype from the SysML profile. The library is organized using SysML *packages* to group related aspects by type. Additionally, SysML specialization relationships are used to order the aspects from most abstract to least abstract.

4.4.3 Establishing Fine-Grain Mappings

To describe how a specific analysis model relates to a component structure model, a *Model Context* is defined. The knowledge captured by fine-grain mappings is encoded in this Model Context. Just like aspects, Model Contexts are also stereotyped with the <<Model Context>> stereotype from the MAsCoM profile making them easy to recognize and computer interpretable. A different model context is needed between every corresponding component and analysis model.

The idea of mapping analysis models to structure models in a specific context was developed previously by Peak *et al.* [4]. They introduced Context Based Analysis Models (CBAM) to bind the parameters of an analysis model to values in a structural model in the context of a specific analysis. If the analysis model is defined to be sufficiently general, it can be reused in multiple contexts. Here, it is recognized that, for a particular component, such bindings between analysis models and structure models often remain the same irrespective of how the component is used within a larger system. It therefore makes sense to establish these bindings at the component level so that the mapping becomes reusable.

Parameter Maps

Model parameters from the component model are linked to parameters of the analysis model using bindings that are captured on a parameter map. These bindings are made using *binding connectors* which are a standard construct of the SysML language. They can be combined with SysML parametrics and constraints to capture algebraic relationships between the parameters. An example of a parameter map is shown in Figure 4.5. In this parameter map, the parameter describing the mass of a structural load is mapped to a corresponding mass parameter in a corresponding analysis model.

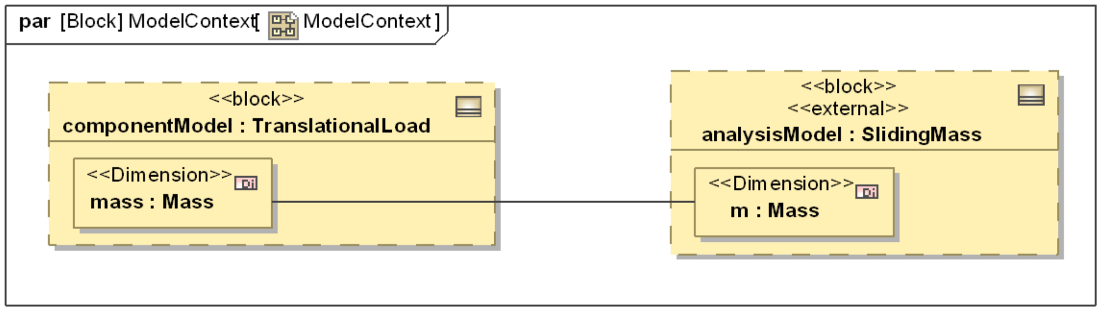

par [Block] ModelContext[ModelContext]
<<block>>
componentModel : TranslationalLoad
<<Dimension>>
mass : Mass
<<block>>
<<external>>
analysisModel : SlidingMass
<<Dimension>>
m : Mass

Figure 4.5: Parameter map between a structural model of a translational load and an analysis model

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 is captured using stereotyped SysML *dependencies*. An example of an interface map is shown in Figure 4.6.

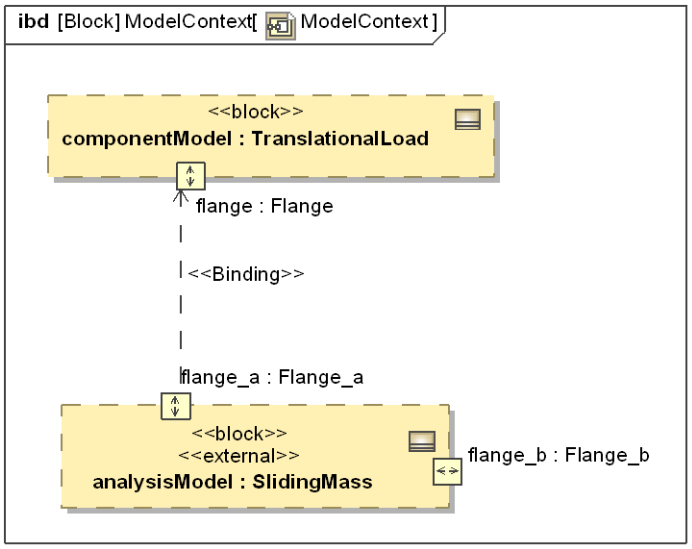

ibd [Block] ModelContext[ModelContext]
<<block>>
componentModel : TranslationalLoad
flange : Flange
<<Binding>>
flange_a : Flange_a
<<block>>
<<external>>
analysisModel : SlidingMass
flange_b : Flange_b

Figure 4.6: Interface map between a structural model of a translational load and an analysis model

4.5 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 along with the knowledge captured within MAsCoMs. The approach relies on the use of graph transformations applied to the structural model to generate an appropriate analysis model.

To simplify this process, the graph transformations use the abstract syntax defined by the domain specific language defined in Section 4.2 to capture these composed system-level analysis models. The abstract syntax of this language is defined by the languages formal metamodel.

4.5.1 Representing the Structural Model and Context

The system-level structural model is represented using SysML. As mentioned previously, the structural model is a system composed of modular component (or subsystem) models. These component models are specializations of models in the component library. To capture this relationship, SysML blocks representing the component models are linked to models in the taxonomy using SysML specialization relationships.

The system-specific component models inherit the appropriate interfaces from the models in the component taxonomy. These models are connected via these appropriate interfaces; these connections are maintained when the corresponding analysis model is generated.

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, an analysis context is used. An analysis context consists of a set of aspects as well as a simulation template. These aspects are the MAsCoM aspects organized in the aspect taxonomy; when the corresponding system-level analysis model is composed; component-level models classified with the appropriate aspects are used.

The simulation template prescribes the simulation parameters and specifies the variables of interest. The simulation template contains the information needed to execute the analysis model such as simulation time and solver information.

4.5.2 Graph Grammar

In order to automatically create an appropriate system-level analysis model, a graph transformations is used to *transform* from a system-level structural model.

This graph grammar is composed of two distinct sets of transformations; the first set maps from the structural model to the domain specific abstract representation of the analysis models. The second set maps from this abstract representation back to a concrete representation of the analysis model in SysML. The first set is described in this section, while the second set is presented in the following section.

The first set of transformations captures the relationship between the system-specific structural models, the appropriate MAsCoMs, and the corresponding analysis model. In part, this first set can be thought of as also capturing the composition relationships present between analysis models.

To simplify the presentation, this first set from the grammar is decomposed into three distinct transformations 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. These three transformations are illustrated in Appendix C.

A triple-graph grammar (TGG) styled approach is taken to defining theses transformations using OMG's Query/View/Transform (QVT) standard. TGGs and QVT standard have been shown to be equally expressive [19]. A correspondence metamodel is used to capture the mapping between the domain specific MOF metamodel, that defines the language for our analysis models, and entities from the SysML metamodel. More precisely, instances of this correspondence metamodel (correspondence graphs) define a mapping between representations of structural models in SysML's concrete syntax and representations of analysis models in our domain specific abstract syntax.

Graph transformations are classically defined using a pre-condition, the part of the graph that is matched, and a post-condition, the replacement graph. The knowledge captured within MAsCoMs provides a component of both the pre-condition and post-condition.

For the system-level transformation, the pre-condition is the structural model and its simulation context along with the appropriate MAsCoM templates. For each component within the structural representation, a matching analysis model is instantiated within the system-level analysis model. The appropriate analysis model is determined by comparing the aspects of the simulation context with the aspects classifying each analysis model. Currently, graph-based pattern matching is also used to compare these two sets of aspects although this is likely not the most efficient implementation.

The component-level transformation insures consistency of component model parameters and interfaces. Therefore the component-level transformation, the interface and parameter maps provide the majority of the information. The appropriate model context has already been selected in the system-level transformation so the necessary interfaces and parameters are generated using the interface and parameter maps as templates. This is first accomplished by replicating the parameters and interfaces of the analysis model in the library. The library models interfaces along with the previously mentioned parameter maps provide the templates for this transformation.

The last transformation is at the connection-level; it generates the connections between interfaces of the component-level analysis models based on the connections between the interfaces of the component models in the structural representation. Currently only a single component-level transformation is defined, but in general a large number are needed to capture the vast differences in connections between different analysis models.

There are several considerations when defining compositions between interfaces. In general, we assume 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. For example, 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. It is likely that such compositions can also be captured in the form of templates and graph transformations similar to the implementation for interface and parameter maps.

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.

4.6 Example: Hydraulic Circuit

In this section, the approach presented in this chapter will be applied to the hydraulic circuit example. A structural representation of the hydraulic circuit is transformed into analysis model. In this case, the analysis model is a Modelica continuous dynamics model. The analysis model is represented in SysML similar to the structural representation; this representation is solver-independent.

4.6.1 Defining the Model Libraries

Defining the structural model library is discussed in Section 3.3.3; it consists of common hydraulic components organized into a taxonomy. The analysis model library contains references into Modelica models that can be composed together and simulated to model the behavior of a hydraulic circuit. The creation of the analysis model library is discussed in more detail in Section 5.4.1.

4.6.2 Creating Model Contexts and Establishing Fine-grain Relationships

Before a composed analysis model can be created, fine-grain relationships must be established between the structural models and analysis models. This is accomplished using Model Contexts along with interface and parameter maps as discussed in Section 4.3.4. For each analysis model of interest, a Model Context is created. Within this Model Context, each analysis model is linked to a corresponding structural component model. Also, each analysis model is related to aspects from the aspect library using dependency relationships. The Model Context for the "ConstantDisplacementPump" analysis model is shown in QQ. The analysis model is related to the structural representation for a Fixed Displacement Pump. In this example, the ConstantDisplacementPump analysis model is labeled with the aspects "Dynamic", "DAE", and "Modelica." This characterizes the analysis model as dealing with the dynamic behavior, being defined using differential-algebraic equation, and in the Modelica language.

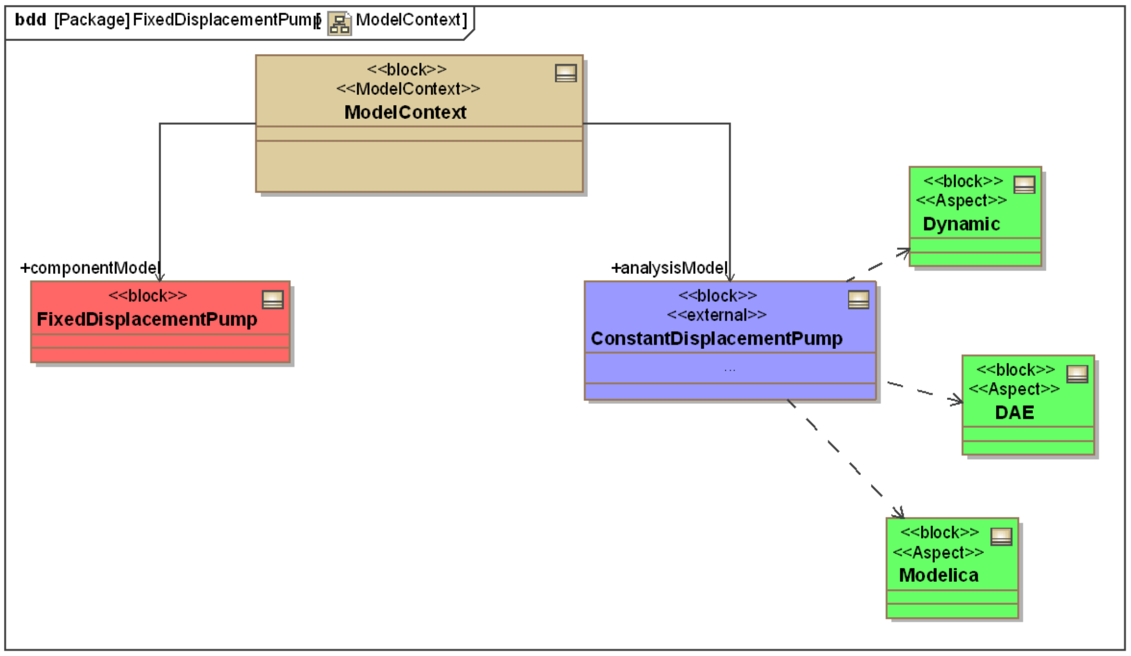


Figure 4.7: Model Context for Constant Displacement Pump Model

After the Model Context has been created, a parameter and interface map is created to capture the fine-grain relationships. The parameter map for the

ConstantDisplacementPump analysis model is shown in Figure 4.5. The “Dconst” parameter of the analysis model is linked to the displacement of the fixed displacement pump. The interface map is shown in Figure 4.6. The interfaces of the analysis model are linked to the interfaces of the fixed displacement pump.

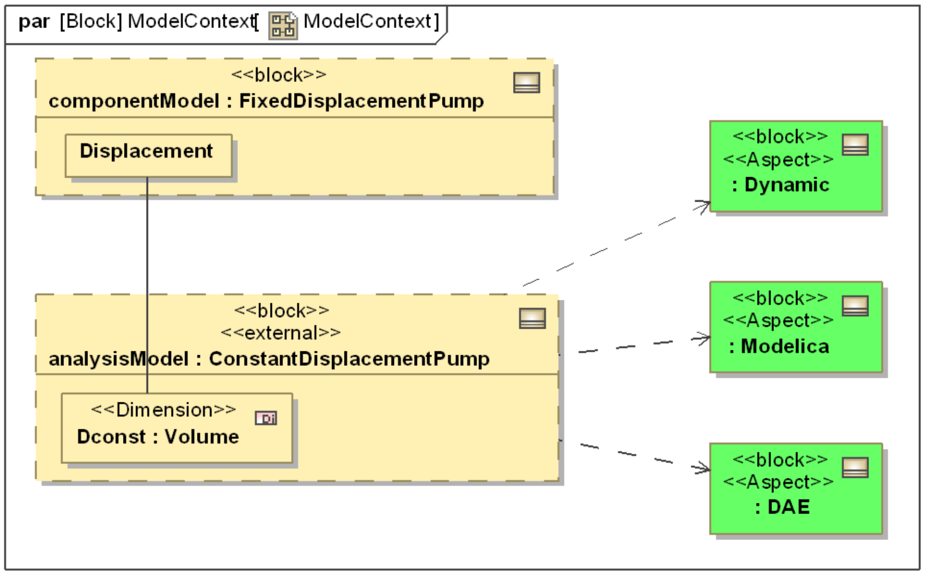


Figure 4.8: Relationship between pump structural model parameter and pump analysis model parameter

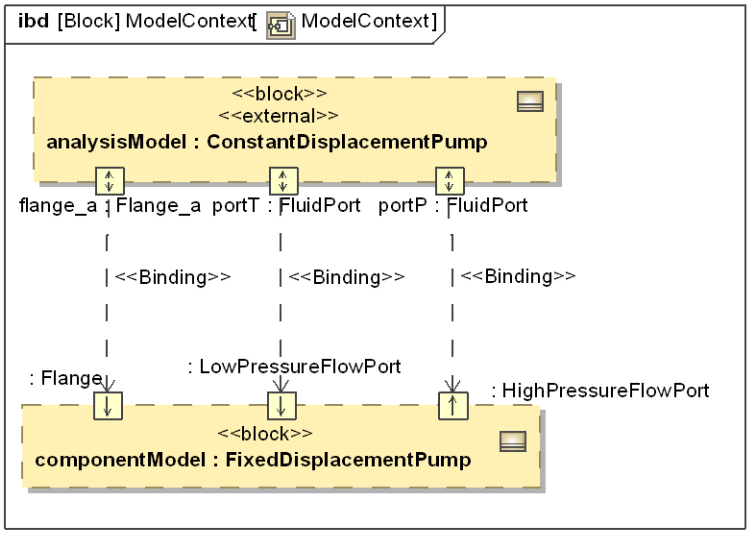


Figure 4.9: Relationship between pump structural model interfaces and pump analysis model interfaces

4.6.3 Structural Model and Context

Once each of the analysis models has been captured in an appropriate Model Context, the captured knowledge is reused to automatically transform from structural representations into analysis models. First, a structural representation of the system needs to be defined. Here, a model of the structural representation of a random hydraulic circuit generated using the synthesis method presented in Chapter 3. This circuit is shown in Figure 4.10. A context is also defined; it captures that the structural model of interest is the hydraulic circuit and the composed analysis model should have certain aspects, namely the “Dynamic”, “Modelica”, and “DAE” aspects.

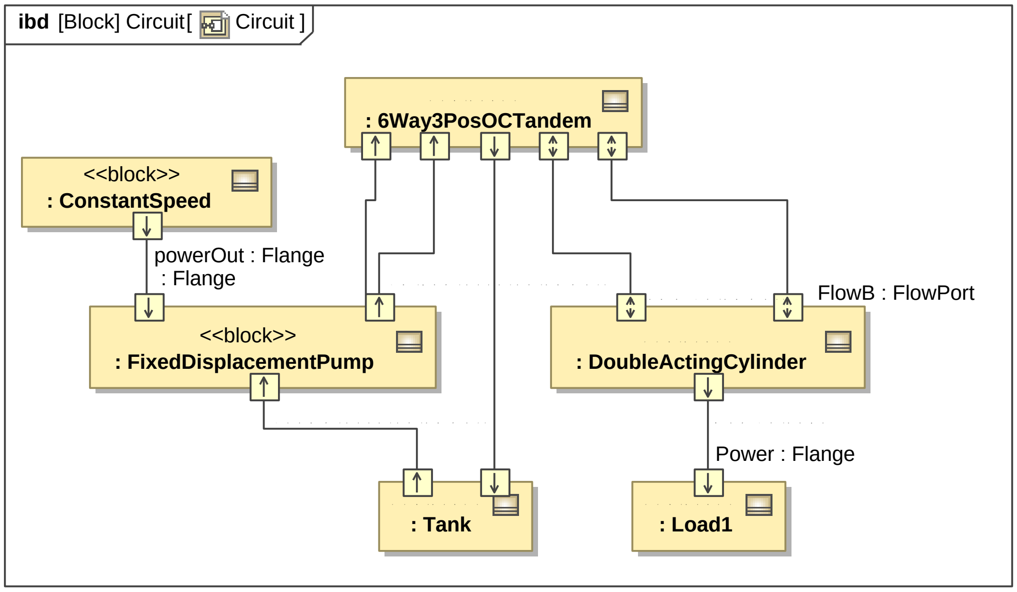


Figure 4.10: Structural model of simple hydraulic circuit.

4.6.4 Composed Analysis Model

Once the structural representation and context have been defined, a composed analysis model is generated. This composed model is shown in Figure 4.11. The composed analysis model has the same layout as the structural representation. All of the structural models of the components have been replaced with appropriate analysis models. Then the interfaces are connected in appropriate fashion. Although not shown, the parameters are also appropriate mapped. This composed analysis model can be transformed into a simulation model and simulation; this will be covered in the next chapter.

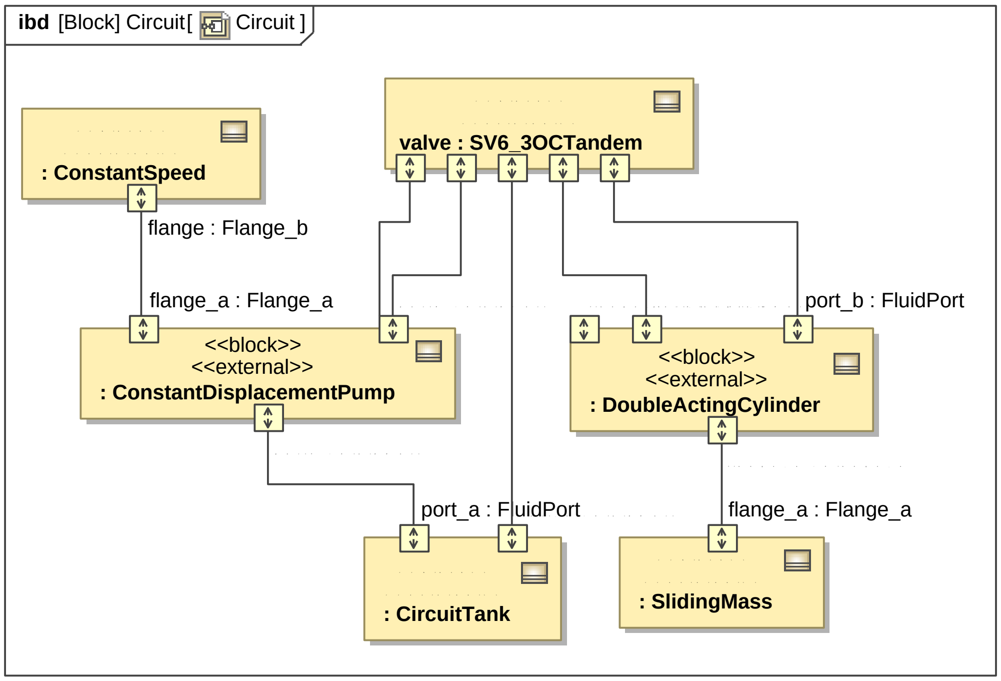


Figure 4.11: Composed analysis model for simple hydraulic circuit

4.7 Discussion

The approach presented uses a graph grammar to capture the composition rules needed to connect together component-level analysis models. Several assumptions are made during this process. Several of these assumptions are implicitly captured within the graph grammar; also the grammar can be extended remove or change some of these assumptions. In the example, assigning causality to the model is left to an analysis or simulation tool. This is not true of all such tools and the graph grammar could be extended to implement a causality assignment algorithm.

A more difficult assumption to relax is that each component of the composed analysis model must be classified with the same aspects. This assumption is valid in the example presented, but for federated analysis models it is not applicable. Federated models may need to be executed by multiple simulation tools. How to capture possible exceptions deserves further investigation. Also, currently the aspect matching algorithm is implemented using simple graph pattern matching. For more complex model libraries, this method will likely become inefficient and improved implementation is worth considering. Also, for federated analysis models an execution manager is required, although several such tools exist (e.g. ModelCenter [33]).

4.8 Summary

In this section, the feasibility of capturing analysis knowledge using DSLs and graph grammars is addressed. The definition a DSL for describing composed-analysis models is described. Reusable model fragments are captured within contained called MAsCoMs and placed into model libraries. Graph grammars are defined to compose analysis models from structural representations. The method is then demonstrated on a simple example involving the generation of an analysis model from a structural representation of a hydraulic circuit

CHAPTER 5

GENERATING SIMULATION MODELS FROM ANALYSIS MODELS[2]

This section describes an approach to generating Modelica simulation models from analysis models in SysML. Modelica simulation models are used as a representative example for simulation models in general. When creating a formal approach for representing continuous dynamics (CD) in SysML, Modelica provides a strong foundation. Modelica has emerged as the language of choice for expressing continuous dynamic system behavior. It is better structured and more expressive than most alternatives such as VHDL-AMS [8] or ACSL [36]. In addition, both SysML and Modelica are similar in that they use base modeling elements that adhere to the principles of object-oriented modeling. Both languages also encourage model reuse through acausal equation-based modeling. Unfortunately, enough differences exist such that a direct one-to-one mapping is not possible. Since SysML is intended to be a general modeling language, some of the specialized semantics of Modelica do not have a direct equivalent in SysML. To overcome these differences, our approach has been to find a good balance between converting some implicit Modelica semantics into explicit constraints in SysML or, when that is not possible, extending the SysML constructs through stereotypes.

[2] Based on work by Tommy Johnson [28]

While SysML is a valuable integration tool, much of that value could be detracted if engineers must manually transform domain-specific models into SysML and vice-versa. In the case of continuous dynamics models, an approach is needed for accomplishing automated, bidirectional transformations between the SysML and Modelica languages.

5.1 Related Work

The need to describe system behavior in terms of equations or constraints has been previously recognized in the work on Constrained Objects (COBs) [48, 49]. COBs provide both a graphical and lexical representation of algebraic relationships that can be used to tie design models to analysis models in a parametric fashion. These COBs recently served as the basis for the development of the SysML parametric diagrams [42]. By establishing a mapping between COBs and SysML, the integration and execution of engineering analyses (such as structural finite element analyses) within the context of SysML has been demonstrated [46]. This section extends this past work on COBs by focusing on the modeling and simulation of the continuous dynamics of systems as defined in Modelica models.

Recently, Fritzson and Pop [50] have worked on the integration of UML/SysML and Modelica to provide support for modeling and simulating continuous dynamics. They have created a UML profile called ModelicaML that enables users to depict a Modelica simulation model graphically alongside UML/SysML information models. The ModelicaML profile reuses several UML and SysML constructs, but also introduces completely new language constructs. Such constructs are the Modelica class diagram, the equation diagram, and the simulation diagram.

Nytsch-Geusen [41] developed a specialized version of UML called UML^H. This version is used in the graphical description and model-based development of hybrid systems in Modelica. The author presents hybrid system models as Modelica models that are based on DAEs combined with discrete state transitions modeled with the Modelica statechart extension. Using a UML^H editor and a Modelica tool that supports code generation, Modelica stubs can be automatically generated from UML^H diagrams so that the user must only insert the equation-based behavior of the system in question. In this chapter, the capabilities of ModelicaML and UML^H are further extended by demonstrating the integration of continuous dynamics models with other SysML constructs for requirements, structure, and design objectives, and by demonstrating the translation between SysML and Modelica through the use of TGGs.

5.2 Domain Specific Language for describing CD Models

In order to transform between the system-level analysis models described in SysML and models that can be simulated described in the Modelica simulation language, a DSL approach is once again taken. This transformation can be viewed as taking models described by different concrete syntaxes but similar abstract syntax. The essence of the two models is the same; from the stand point of a domain-specific language both capture the same pieces of knowledge.

By explicitly capturing the mapping between the concrete and abstract syntaxes, model-driven software development methods are used to simplify the creation of computer-code to execute the transformation. The same approach as in Section x is taken to map between the concrete syntax in SysML into an abstract syntax defined by the explicit metamodel. A tool integrator is then implemented to generate code in the Modelica textual language. This step would not be necessary if a Modelica tool was capable of simply executing the abstract syntax.

5.2.1 Abstract Syntax

To define the abstract syntax of this domain specific language, the Modelica metamodel is formally and explicitly defined using assumptions about the structure of the implicitly implemented metamodel from Modelica tools as a guide. (In this thesis, the Modelica tool of chose is Dymola [10]). For the purposes of simply demonstrating the feasibility and applicability of the presented method, the explicitly defined metamodel remains fairly simple and does not exhaustively cover every construct in the Modelica language. This metamodel is once against specified in MOF; a simplified visual illustration is shown in Figure 5.1.

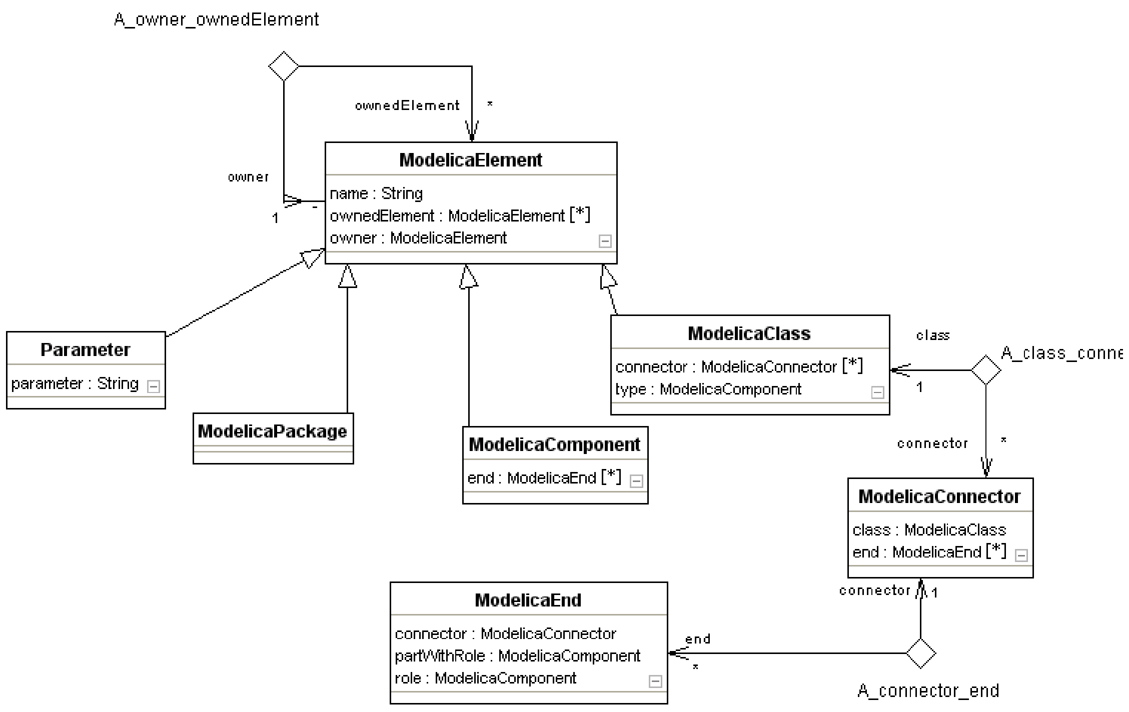


Figure 5.1: Simplified Modelica metamodel represented in MOF.

5.2.2 Concrete Syntax

As in the previously defined DSLs, a concrete syntax is needed to completely specify the DSL. In this case, there are two separate concrete syntaxes: one defined using the graphical constructs of SysML as a foundation and the other being the Modelica textual language. The Modelica language is specifically designed for representing continuous dynamics models so a clear mapping exists between it and the defined abstract syntax. This is not the case for SysML because SysML is a more general purpose language.

5.2.3 Representation of Continuous Dynamics models in SysML

When defining the concrete syntax for representing continuous dynamics in SysML, Modelica is used as the foundation because of its well defined structure and sue of object-oriented modeling concepts. Although there is argument over exactly which SysML constructs best fit the description of continuous dynamics systems, blocks are chosen here to represent Modelica models as in [26]. SysML ports are used to describe the interfaces of the model and SysML value properties

Because the DSL approach facilitates the formal modeling of the mapping between the abstract syntax and possible concrete syntaxes, the particular chose of SysML constructs is unambiguously defined as well as easy to adjust.

Johnson also shows how the majority of the constructs present in Modelica can be analogously represented in SysML to allow for the creation of fully detailed “white box” continuous dynamics models as well as “black box” models which act as references for existing, external Modelica models. The approach of using “black box” models is taken here because it is sufficient for the examples presented. In the “black box” approach models in SysML that relate to fully specified models defined using the textual Modelica language. These models can be thought of as pre-specified library models which are a common feature of most domain-specific simulation tools.

5.3 Transforming Between SysML and Modelica Models

Many methods exist for implementing transformations between various modeling languages such as the use of the QVT or TGG standard as mentioned in Section 2.2. An approach similar to Section 3.1.2 is taken here:

1. The abstract syntax of the domain specific language is captured in a MOF-compliant metamodeling tool as described in the previous section. MOFLON is used as the meta-modeling tool because of its code generation capabilities.
2. A SysML profile is defined within the SysML modeling tool. In this case, the profile is specifically designed to facilitate the representation of "black box" models in SysML. This profile is shown in Figure 5.2. Stereotypes are also defined to capture references to a particular external model library.
3. MOFLON is used to generate Java Metadata Interface (JMI) based code that implements the metamodel.
4. Query/View/Transformation (QVT) based transformation rules are also defined in MOFLON to map between the stereotyped SysML profile and a specific instance of the metamodel. This serves the role of a translator or compiler between the concrete syntax and the abstract syntax.
5. MOFLON is used to generate JMI code that implements these transformations.

6. The code generated by MOFLON is combined with a JMI-compliant SysML tool. This extends the tool to provide the capability of authoring models defined by the DSL.
7. A tool integrator is implemented to create Modelica textual code from the abstract syntax.

A SysML model is stereotyped using the profile. It is then translated into an abstract representation by executing the JMI code. This abstract representation is an abstract syntax graph; this graph is the abstract representation of the defined model.

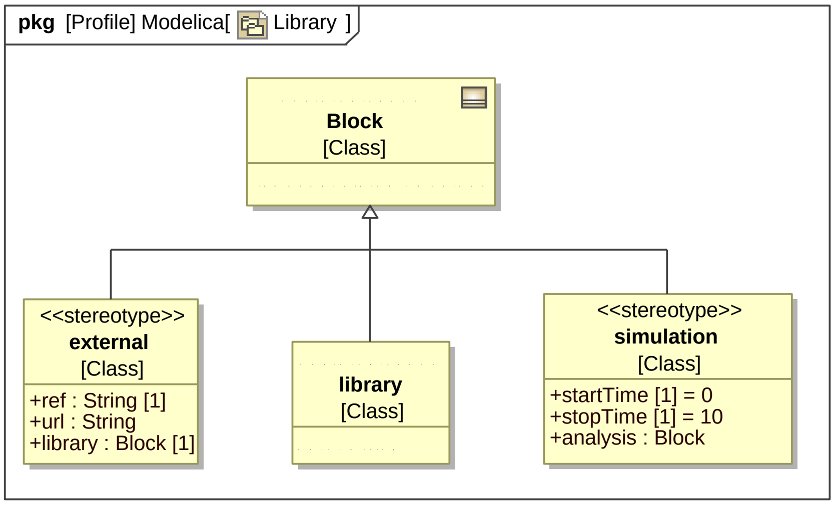


Figure 5.2: Profile for capturing "Black Box" models

5.4 Example: Hydraulic Circuit

This section describes the creation of a simulation model that can be compiled and executed by a Modelica simulation tool such as Dymola.

5.4.1 Referencing models in a model library

As described earlier, useful Modelica models are captured in a model library described in SysML. Each model in the model library is a "black box" model; it references an existing model outside of the SysML tool. In order to create a "black box" model and therefore reference an external model, several pieces of information are needed. These are captured within the <<Library>> and <<External>> stereotypes. The <<Library>> stereotype requires the "url" tag where information pointing to the location of the library is stored. The <<External>> 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.

A SysML block representing the fluid power library and a SysML block representing the "ConstantDisplacementPump" model is shown in Figure 5.3. The fluid power library block has a "url" tag pointing to the location of the library. The "ConstantDisplacementPump" model uses the "ref" tag to describe the location of that model within the fluid power library.

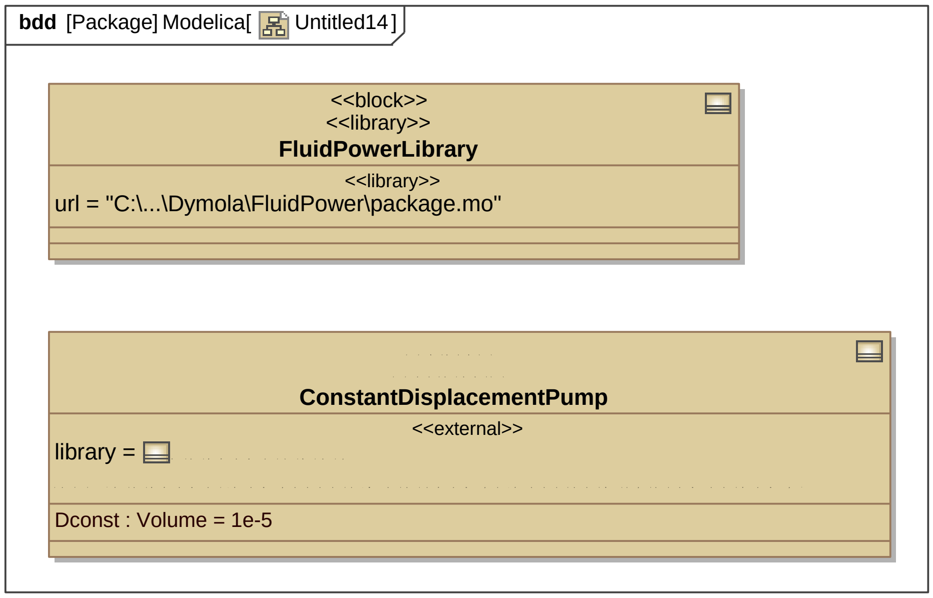


Figure 5.3: Pump model from library along with abstract model of the library

5.4.2 Generating Modelica code from an Analysis Model

The composed analysis model used as an example is the one created in the previous chapter. This model is shown in Figure 5.4. To create a simulation model from this model, some additional knowledge is required. In this case, because it is a dynamic simulation, the start and stop time is required. This is captured in a SysML block modeling the simulation. This is shown in Figure 5.5. The resulting code is shown in Figure 5.6.

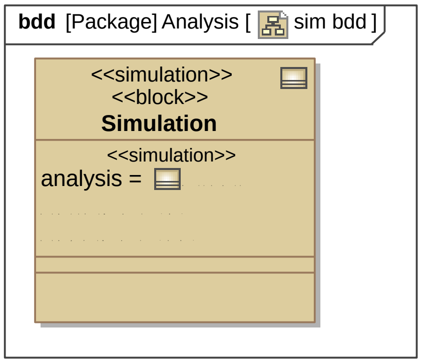


Figure 5.4: Simulation model

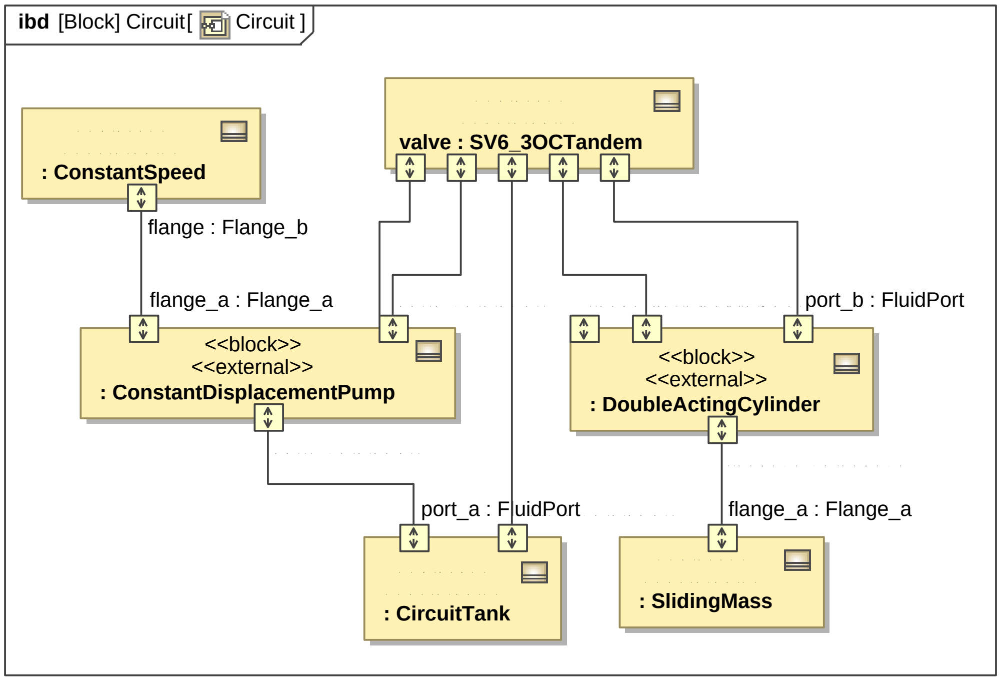
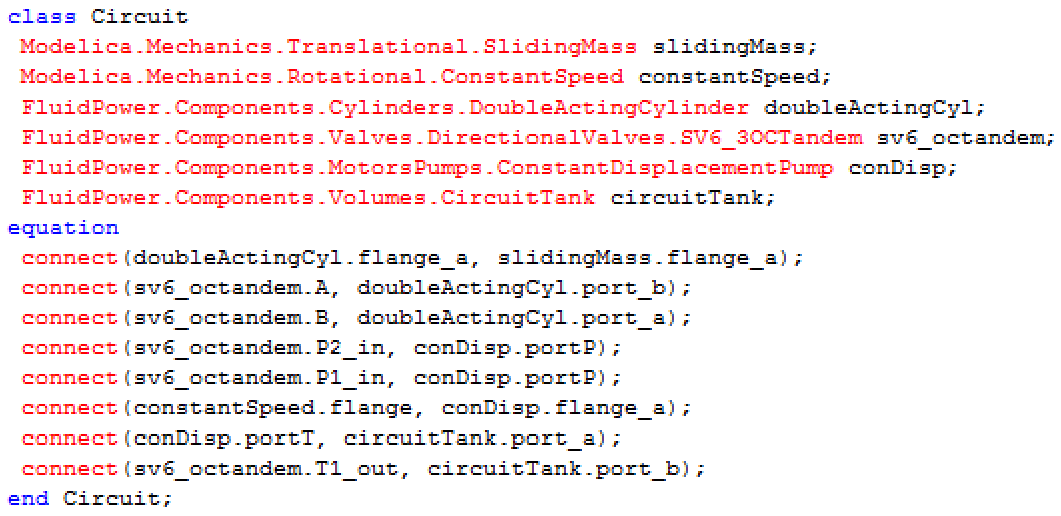


Figure 5.5: Composed analysis model



```
class Circuit
 Modelica.Mechanics.Translational.SlidingMass slidingMass;
 Modelica.Mechanics.Rotational.ConstantSpeed constantSpeed;
 FluidPower.Components.Cylinders.DoubleActingCylinder doubleActingCyl;
 FluidPower.Components.Valves.DirectionalValves.SV6_3OCTandem sv6_octandem;
 FluidPower.Components.MotorsPumps.ConstantDisplacementPump conDisp;
 FluidPower.Components.Volumes.CircuitTank circuitTank;
equation
 connect(doubleActingCyl.flange_a, slidingMass.flange_a);
 connect(sv6_octandem.A, doubleActingCyl.port_b);
 connect(sv6_octandem.B, doubleActingCyl.port_a);
 connect(sv6_octandem.P2_in, conDisp.portP);
 connect(sv6_octandem.P1_in, conDisp.portP);
 connect(constantSpeed.flange, conDisp.flange_a);
 connect(conDisp.portT, circuitTank.port_a);
 connect(sv6_octandem.T1_out, circuitTank.port_b);
end Circuit;
```

Figure 5.6: Code generated from composed model

5.5 Summary

In this section, the feasibility of capturing creating simulation models from analysis models is addressed. Modelica is used as the representative language. A DSL is defined to capture the simulation models. Reusable model fragments are referenced from external libraries. The method is demonstrated with the generation of Modelica code from an analysis model of a hydraulic circuit in SysML.

CHAPTER 6

DISCUSSION AND CLOSURE

In this thesis, the Model-Driven Software Development concepts of formal DSLs and model transformations are applied to the capture of design knowledge. This final chapter reviews the overall approach, discusses limitations, and highlights possible future work.

6.1 Review and Evaluation

The motivation behind this thesis is an open-ended question concerning the efficient representation of design knowledge. Throughout this thesis, concepts from Model-Driven Software Development, mainly the use of domain specific languages and graph-based model transformations, have been used to capture various pieces of design knowledge. The examples of design knowledge used throughout are representative; therefore it is likely that the prescribed approach can be applied to a wider range of problems (not just the design of toy examples or hydraulic circuits). But from the work presented here, it would be bold to claim that all types of design knowledge could be formally captured in this manner.

The use of formal models represented using formal domain specific languages throughout the design process promotes traceability, transparency, consistency, and automated transformation. The presented approach facilitates the definition of domain specific languages and therefore likely better enables designers to apply MBSE to complex systems. The major problem with the work presented in this thesis, however, is that it has not been tested on the target audience: systems and disciplinary engineers working in a variety of domains. One can assume that through improvement of implementation details this approach to specify DSLs could be valuable for the target audience; however, that value has yet to be confirmed.

The effectiveness of the presented approach in capturing the prescribed design knowledge about the example problem is encouraging. Although there are clearly limitations, as discussed in each chapter and further addressed in the next section, none seem to be the results of an inherent and fundamental flaw in the approach. Therefore, it seems reasonable to claim that using such an approach to capture design knowledge is generally feasible.

6.2 Limitations

The limitations of the presented approaches to capturing specific design knowledge have been presented in each individual chapter. This section discusses high-level limitations to capturing design knowledge in general using the presented approach.

Expressivity of the Metamodels

Throughout this thesis, the metamodeling language used is OMG's MOF; MOF is designed to be an effective meta-language for models that are inherently object-oriented or are based on object-oriented principles. The DSLs introduced throughout have been of an object-oriented nature, but this may not be the case to capture knowledge in certain domains. But since the trend in systems engineering is towards modularity and other object-oriented concepts in designed systems, a majority of languages for describing aspects of these systems are also objected-oriented in nature.

Ease of Using Graph Grammars

Graph grammars are used throughout this work to capture knowledge. One weakness of this approach is that some of the knowledge being captured within the graph grammars is implicit. A particular modification rule might be designed to insure a certain component is also connected to another component, but it does not explicitly capture, for example, whether these two components must *always* be connected. Also, creating transformations rules that implicitly capture certain knowledge can become tedious and difficult. How this complexity presents itself deserves further consideration, although graph grammars have been used for a wide variety of applications as mentioned throughout the thesis.

Fidelity/Abstraction

Also, throughout this thesis models are assumed to be at an “appropriate” level of abstraction or fidelity when being composed. It is also not clear if is possible to rate a model’s “fidelity” or “level of abstraction” using an absolute and unambiguous scale.

Scalability

Applying graph transformations to increasingly complex systems models can become very computational expensive. There have been a number of case-studies using graph transformations applied to very complex software systems, and in this thesis this computational expense never presented a problem.

6.3 Future Work

Obviously, it would be prudent for future work to focus on addressing the limitations presented in the previous section. Also, the work presented here has only attempted to establish the feasibility of using the presented methods .One obvious extension is the comparison of the work presented here with other approaches to capture design knowledge. Such a rigorous comparison is likely to shed more light on the question of how *should* design knowledge be captured, versus simply how it *can* be captured.

Throughout the work presented in this thesis, only a single application domain is considered. Transformations are used to transformation from one DSL to another, but interactions between models represented using different DSLs is not considered. Also, interactions with other domains are largely abstracted. For example, the interaction between the hydraulic circuit and the corresponding mechanical structure is significantly simplified. With the DSL approach, it is likely that both of these domains would be described using different DSLs. How models represented using such DSLs would interact deserves consideration.

Also, completing the loop shown in the high-level view on Figure 1.1 by using an optimization algorithm is left for future work. The use of an evolutionary program with the synthesis approach is demonstrated in this thesis but to truly complete the loop an attribute grammar [53] or similar method is needed to provide appropriate initial sizing to the components. Else, the applied optimization algorithm may generate hydraulic circuits with very poor parameters which become difficult to simulate.

APPENDIX A

GLOSSARY OF TERMS

Abstract syntax – describes the “essence” of the model; the abstract syntax representation is independent of any particular concrete representation

Concrete syntax – describes how a model can be represented concretely. For example, with programming languages the concrete syntax includes punctuation, etc. that is not included in the abstract syntax. A concrete syntax can be either textual or visual.

Domain-Specific language – a language specifically designed for describing a particular problem domain. Defined by an abstract syntax as well as at least one concrete syntax. In general, a domain-specific language is mapped to a specific domain to give it semantic meaning.

Graph – A collection of nodes and edges. For the purpose of this thesis, the nodes and edges are generally labeled. Also, the edges are directed.

Metalanguage – a language for describing a metamodel, just as a metamodel describes a model.

Metamodel – language for defining models, a metamodel provides the available constructs and relationships that can be used to describe a model. A particular model is an instance of its metamodel.

Meta-circularity – the use of a metalanguage to define itself. This allows the practical definition of metalanguages.

Model-Based Systems Engineering – The use of models instead of documents to describe all aspects of the systems engineering process.

Model-Driven Architecture – Pre-cursor to Model-Driven Software Development in computer science; model-driven/based architecture relies on

Model-Driven Software Development – From computer science, models are used to automate the generation of code. This is a shift from the more conventional approach of using models to constitute documentation.

Modeling Language – any language that can be used to express information or knowledge in a structure that is defined by consistent set of rules.

Profile – A light weight extension mechanism that SysML shares with UML; a profile can be used to quickly extend either UML's or SysML's metamodel.

SysML – Object Management Group's Systems Modeling Language. It is a standardized general-purpose visual modeling language for systems engineering.

Syntax – The rules and principles that govern the structure of a language

Semantics – the meaning of a language

UML – Object Management Group's Unified Modeling Language. It is a standardized general-purpose visual modeling language in the field of software engineering.

APPENDIX B

SYNTHESIS GRAMMAR

The graph grammar used to create design alternatives for the hydraulic circuit example is presented in this appendix. As mentioned in Section 3.3.2, these transformations reflect actions designers might take to create a hydraulic system. The presented graph transformations:

- Select a random component from the model library
- Add an instance of a cylinder to the circuit and configure it to actuate a load.
- Add an instance of a directional valve to the circuit to control a cylinder.
- Add an instance of a pump to the circuit to provide flow to the directional valve.
- Add an instance of a tank to the circuit to provide flow to instances of pumps.

These transformations are implemented using MOFLON and executed using an order determined by traversing the decision graph.

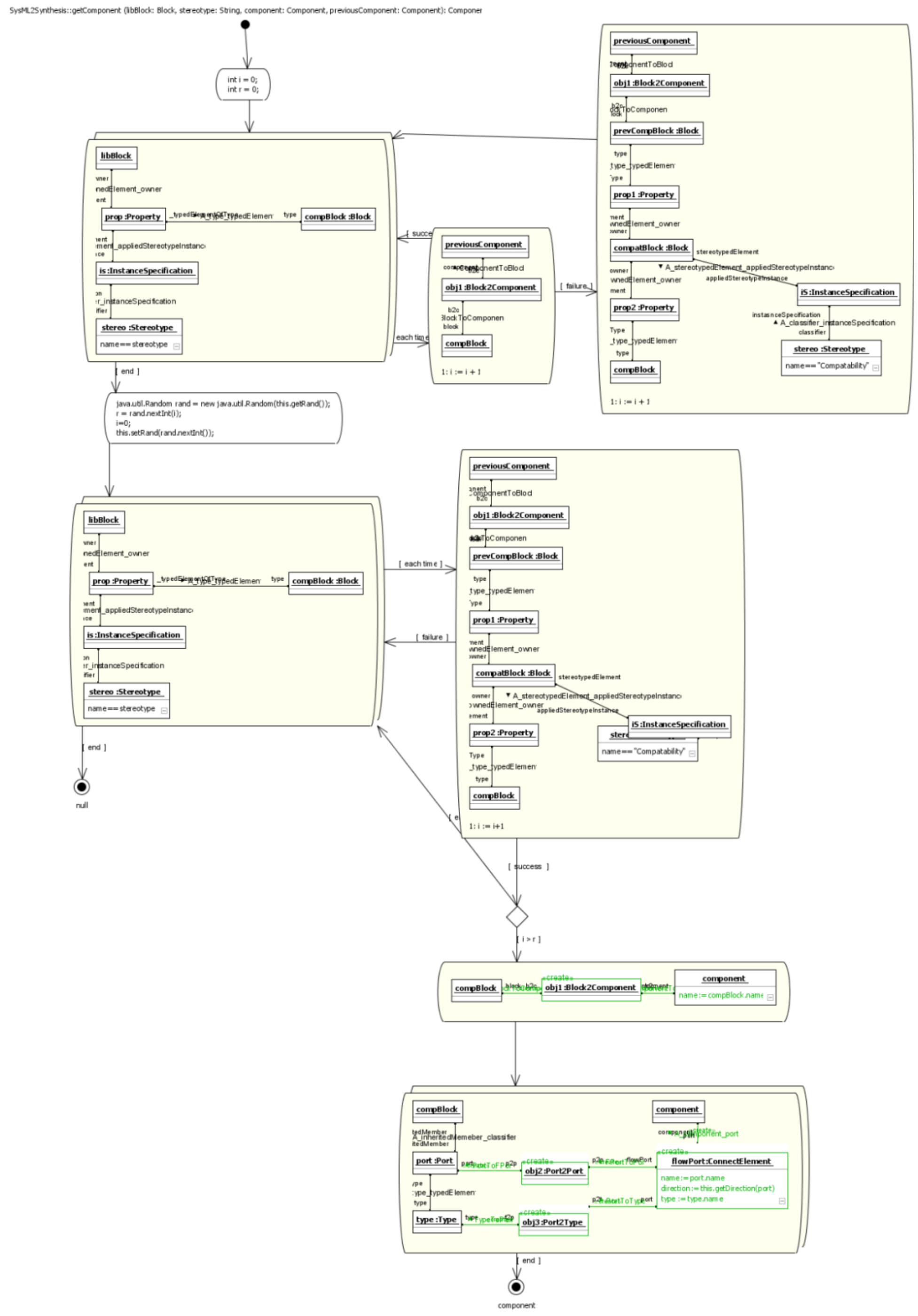


Figure B.1: Graph pattern for matching random component models in model library

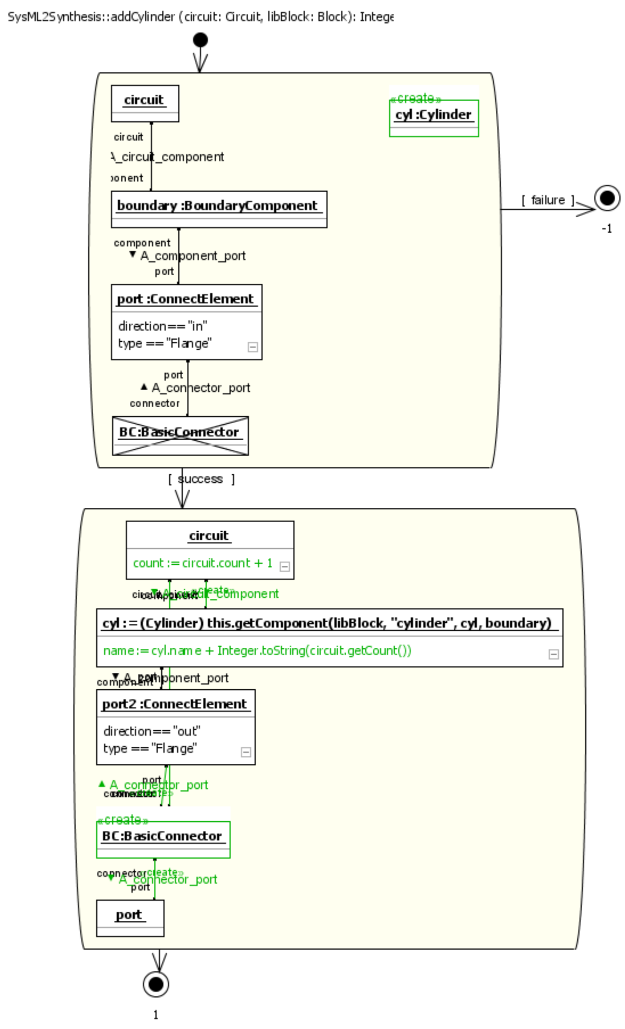


Figure B.2: Graph transformation for adding a cylinder to the hydraulic circuit

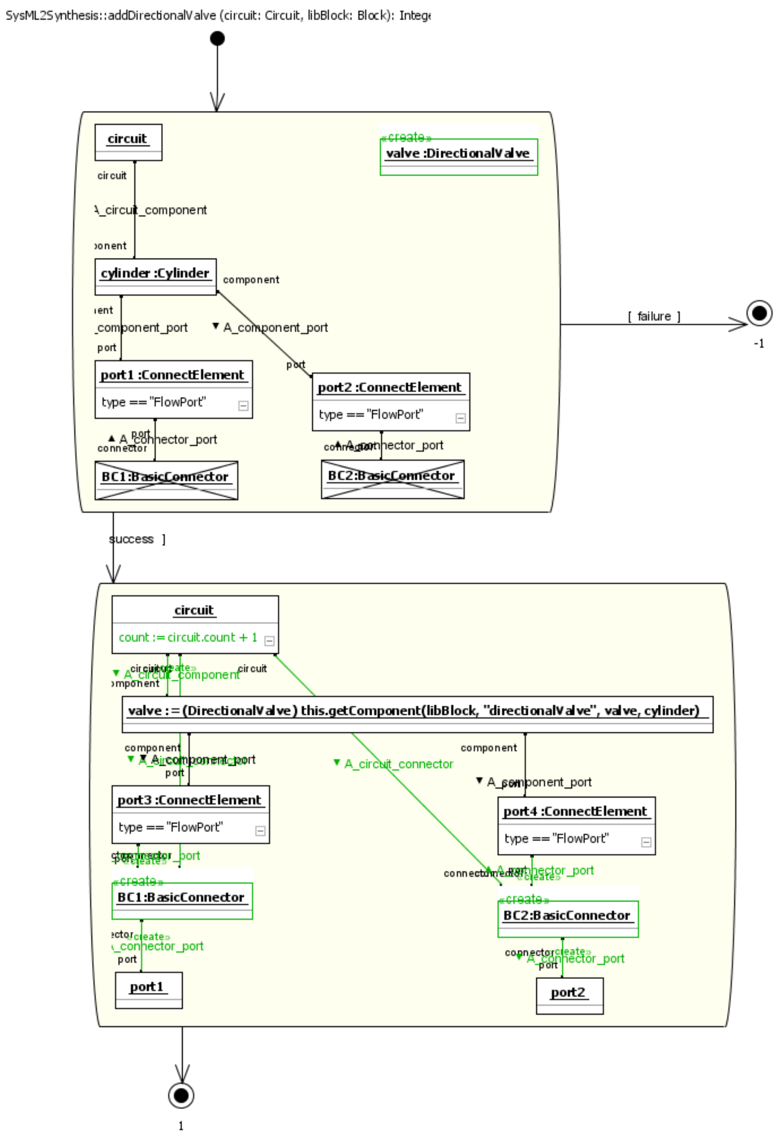


Figure B.3: Graph transformation for adding a directional valve to the hydraulic circuit

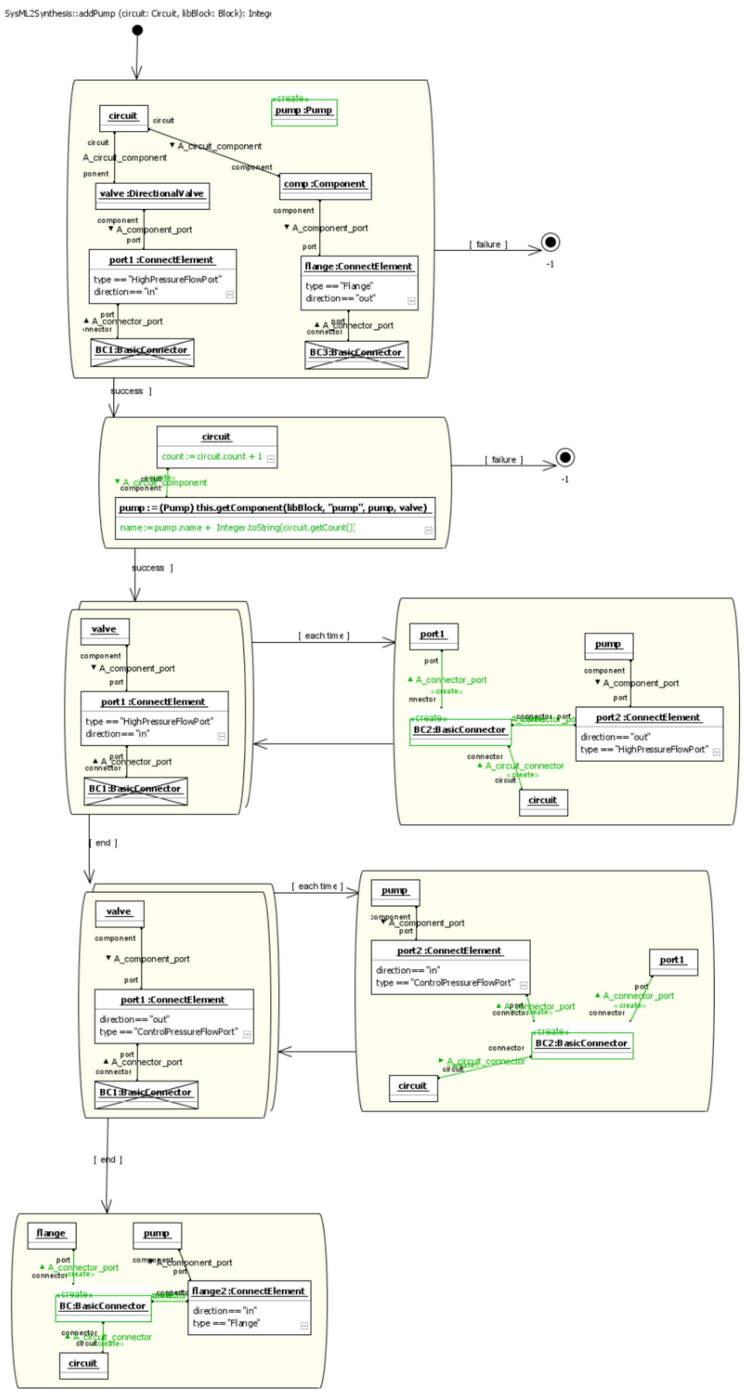
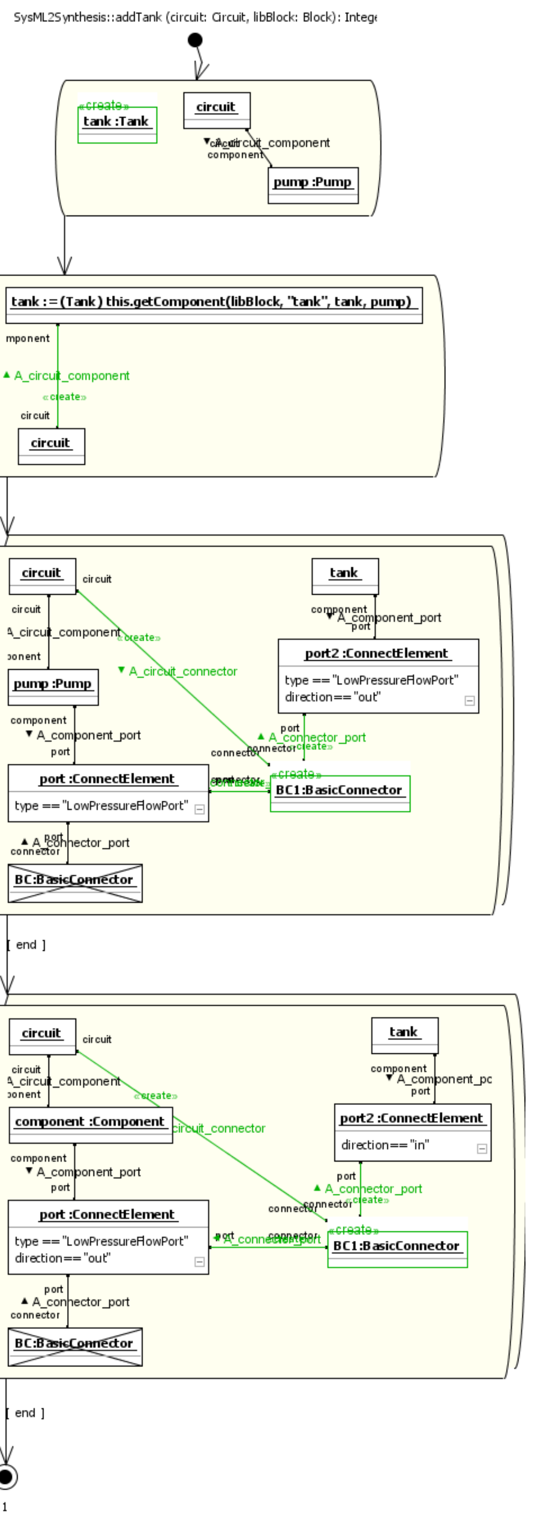


Figure B.4: Graph transformation for adding a pump to the hydraulic circuit



SysML2Synthesis::addTank (circuit: Circuit, libBlock: Block): Integer
«create»
tank :Tank
circuit
A_circuit_component
pump :Pump
tank := (Tank) this.getComponent(libBlock, "tank", tank, pump)
A_circuit_component
«create»
circuit
port2 :ConnectElement
type == "LowPressureFlowPort"
direction == "out"
A_circuit_connector
A_component_port
A_connector_port
port :ConnectElement
BC1:BasicConnector
BC:BasicConnector
[end]
component :Component
direction == "in"
1

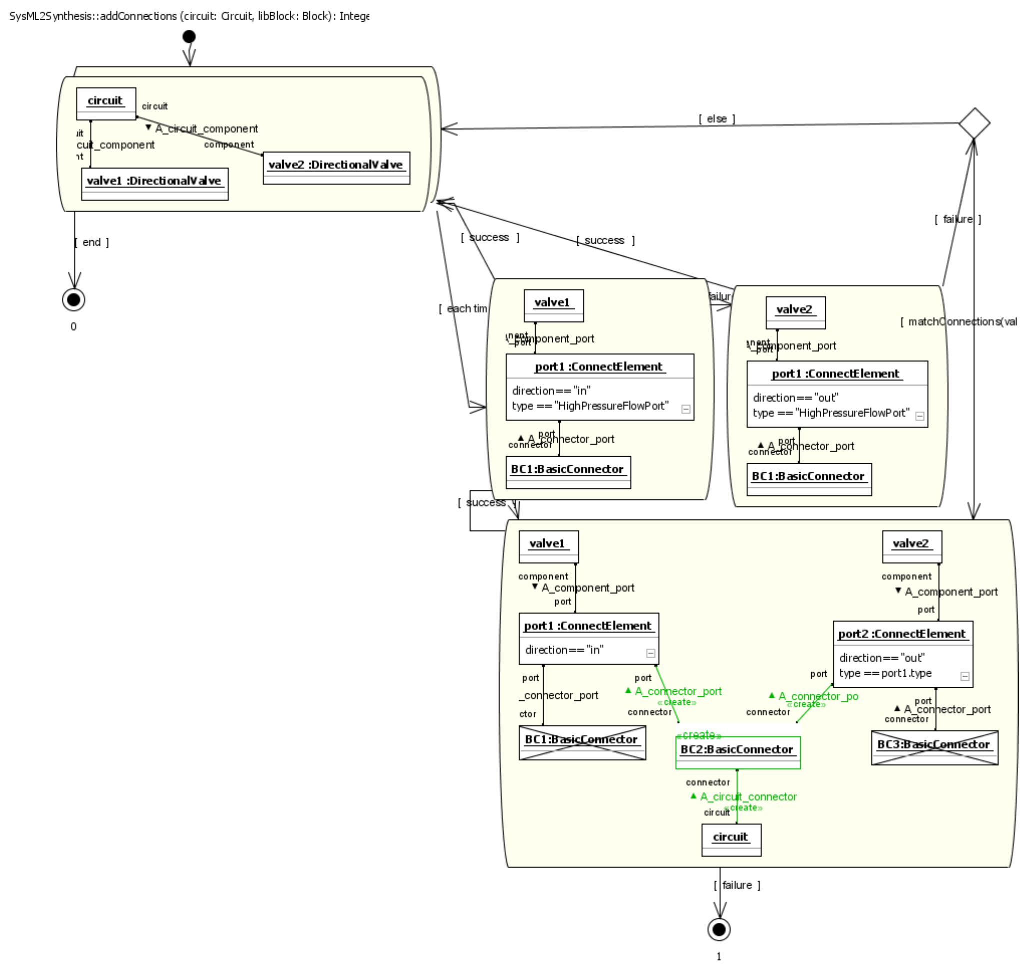


Figure B.6: Graph transformation for connecting similar directional valves

APPENDIX C

ANALYSIS MODEL COMPOSITION GRAMMAR

The three graph transformations used to create a composed analysis model from a structural representation are presented in this appendix. 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.

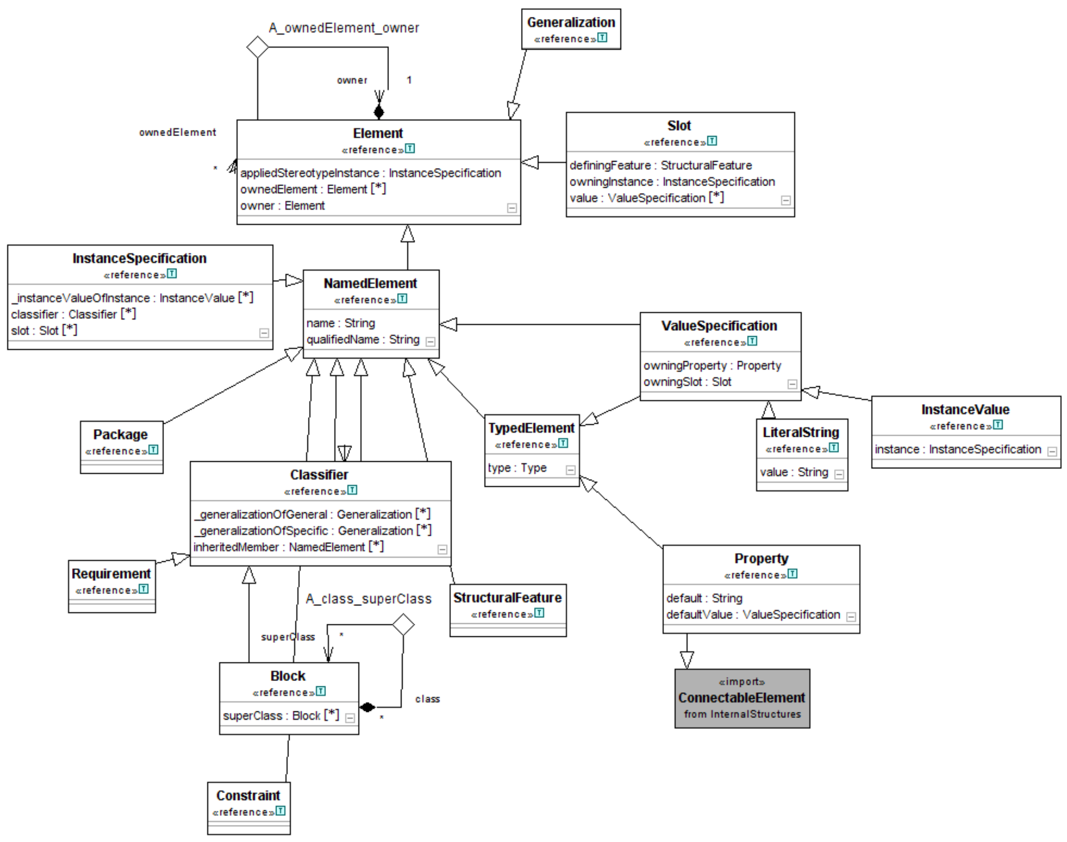


Figure C.1: Partial SysML metamodel used when defining transformations

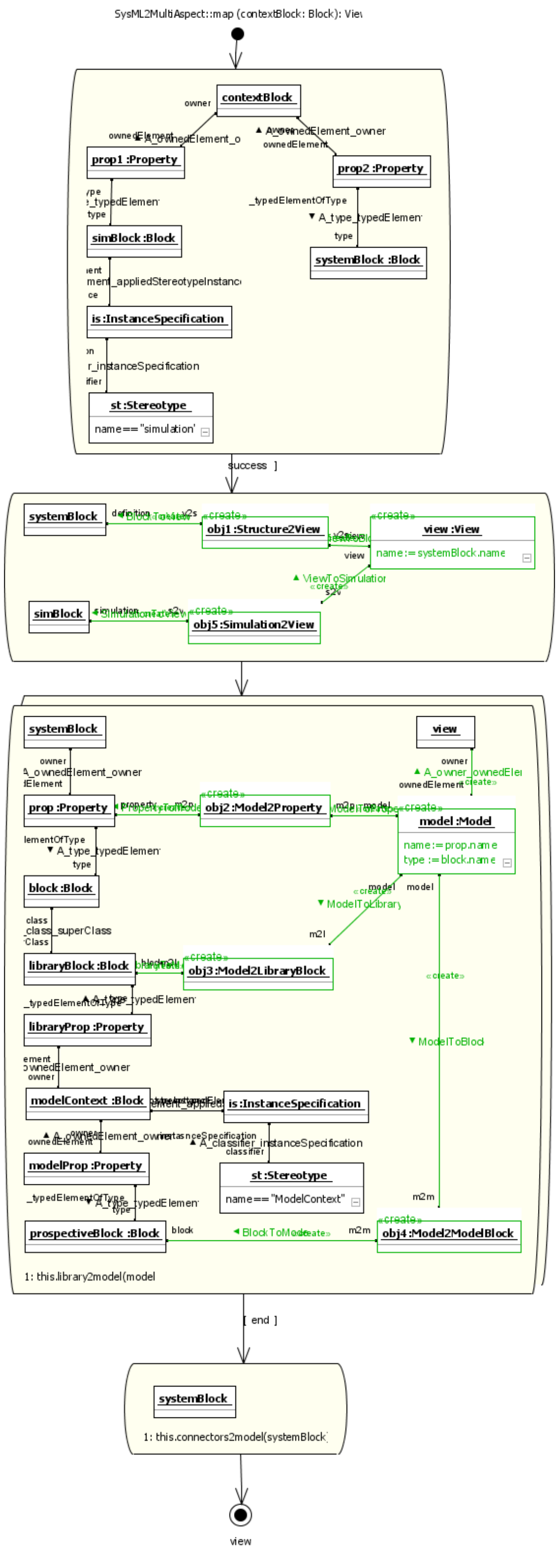


Figure C.2: System level transformation

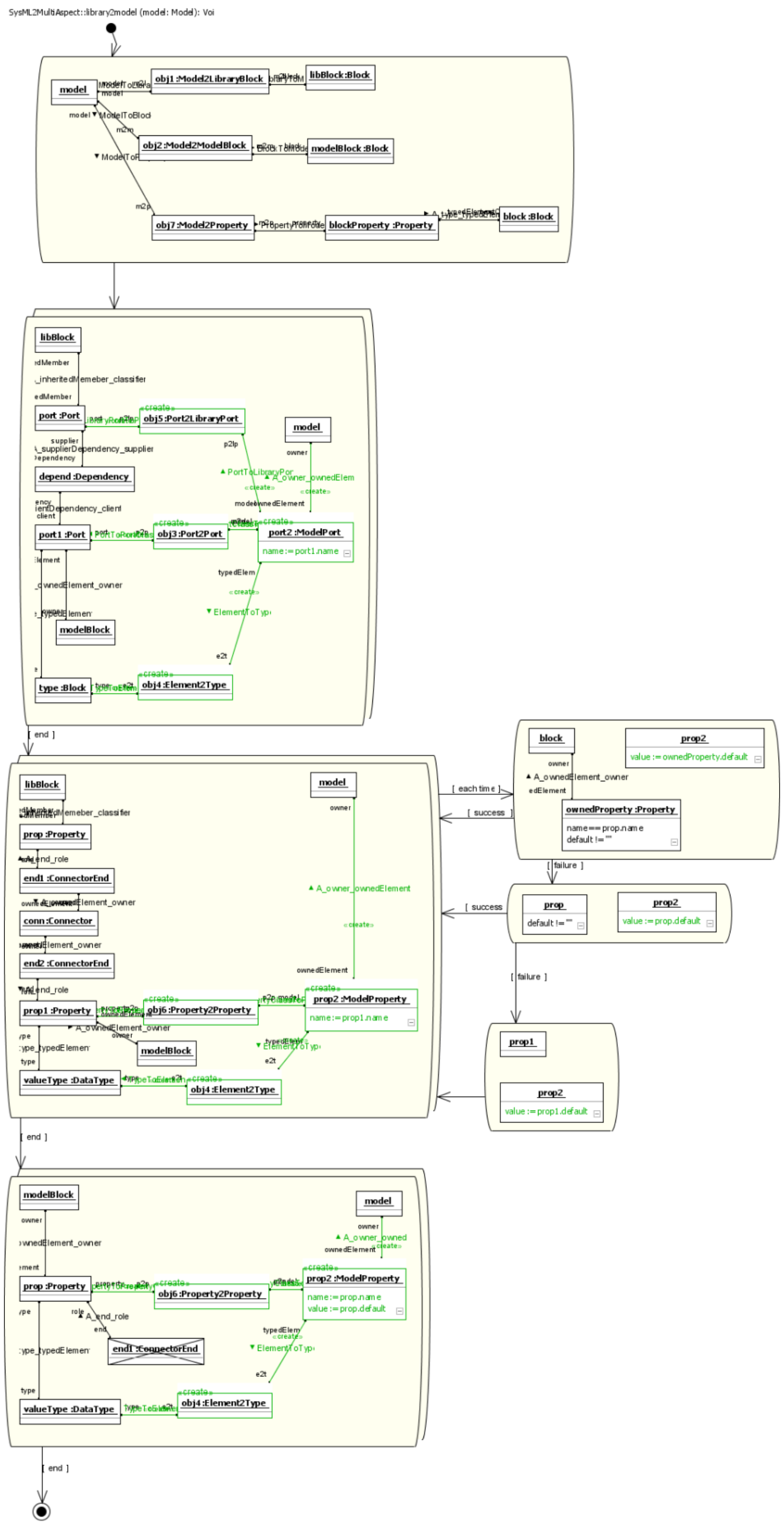


Figure C.3: Component-level transformation

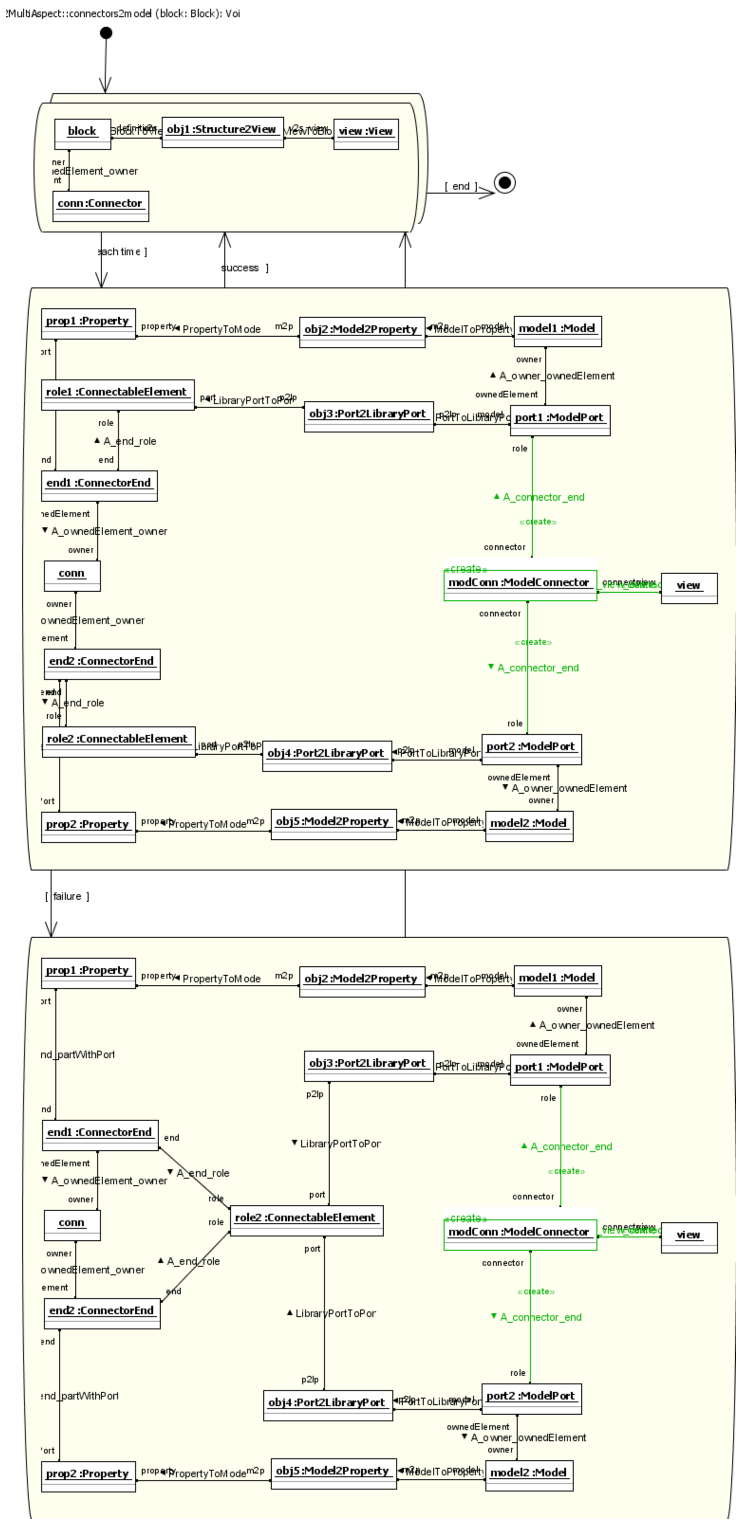


Figure C.4: Connection-level transformation

REFERENCES

[1] Alber, R., Rudolph, S., and Kröplin, B., 2002, "On Formal Languages in Design Generation and Evolution," *Fifth World Congress on Computational Mechanics*, Vienna, Austria.

[2] 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.

[3] Bajaj, M., Peak, R. S., and Paredis, C. J. J., 2007, "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*, ASME, Las Vegas, Nevada, USA.

[4] Bajaj, M., Peak, R. S., and Paredis, C. J. J., 2007, "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*, ASME, Las Vegas, Nevada, USA.

[5] Baldwin, C. Y., and Clark, K. B., 1999, *Design Rules: Volume 1. The Power of Modularity*, The MIT Press.

[6] Bolognini, F., Seshia, A. A., and Shea, A. K., 2007, "A Computational Design Synthesis Method for Mems Using COMSOL," *COMSOL Users Conference*.

[7] Campbell, M. I., Cagan, J., and Kotovsky, K., 1999, "Agent-Based Synthesis of Electro-Mechanical Design Configurations," *ASME Journal of Mechanical Design*, **122**(1), pp. 61-69.

[8] 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.

[9] Czarnecki, K., and Helsen, S., 2003, "Classification of Model Transformation Approaches," *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*

[10] Elmqvist, H., Brück, D., and Otter, M., 1995, "Dymola-User's Manual," *Dynasim AB, Research Park Ideon, Lund, Sweden*.

[11] 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.

[12] Eppinger, S. D., Sosa, M. E., and Rowles, C. M., 2000, "Designing Modular and Integrative Systems," *ASME 2000 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Baltimore, Maryland, USA.

[13] Evans, E., 2004, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley Professional.

[14] 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).

[15] Fisher, J., 1998, "Model-Based Systems Engineering: A New Paradigm," in *INCOSE Insight*, vol. 1.

[16] 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.

[17] Giese, H., Levendovszky, T., and Vangheluwe, H., 2007, "Summary of the Workshop on Multi-Paradigm Modeling: Concepts and Tools," *Lecture Notes In Computer Science*, **4364**, pp. 252.

[18] Goldberg, D. E., 1989, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

[19] Greenyer, J., Kindler, E., 2007, "Reconciling Tggs with Qvt," in *Model Driven Engineering Languages and Systems, MoDELS 2007*, Springer, Berlin / Heidelberg.

[20] 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.

[21] Holland, J. H., 1992, *Adaptation in Natural and Artificial Systems*, MIT Press Cambridge, MA, USA.

[22] 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.

[23] ISO/IEC, 2005, Unified Modeling Language Specification, http://www.omg.org/cgi-bin/apps/doc?formal/05-04-01.pdf. March 10, 2009.

[24] Jackson, D., 2002, "Alloy: A Lightweight Object Modelling Notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **11**(2), pp. 256-290.

[25] Jobe, J. M., 2008, *Multi-Aspect Component Models: Enabling the Reuse of Engineering Analysis Models in SysML*, Masters Thesis, Department of Mechanical Engineering, Georgia Institute of Technology, Atlanta, GA.

[26] 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.

[27] Johnson, T. A., Paredis, C. J. J., Burkhart, R. and Jobe, J. M., 2007, "Modeling Continuous System Dynamics in SysML," in *2007 ASME International Mechanical Engineering Congress and Exposition*, ASME, Seattle, WA.

[28] Johnson, T. J., 2008, *Integrating Models and Simulations of Continuous Dynamic System Behavior into SysML*, Masters Thesis, G. W. Woodruff School of Mechanical Engineering, Georgia Insitute of Technology, Atlanta, GA.

[29] Königs, A., and Schürr, A., 2005, *Multi-Domain Integration with MOF and Extended Triple Graph Grammars*, Internat. Begegnungs-und Forschungszentrum für Informatik.

[30] Kopena, J. B., and Regli, W. C., 2003, "Functional Modeling of Engineering Designs for the Semantic Web," *Data Engineering*, **26**(4), pp. 55-61.

[31] Koza, J. R., Bennett III, F. H., Andre, D., Keane, M. A., and Dunlap, F., 1997, "Automated Synthesis of Analog Electrical Circuits by Means of Genetic Programming," *IEEE Transactions on Evolutionary Computation*, **1**(2), pp. 109-128.

[32] Lee, C. Y., Ma, L., and Antonsson, E. K., 2001, "Evolutionary and Adaptive Synthesis Methods," *Formal Engineering Design Synthesis*, Cambridge University Press, pp. 270-320.

[33] Malone, B., and Papay, M., 1999, "ModelCenter: An Integration Environment for Simulation Based Design," *Simulation Interoperability Workshop*.

[34] Mellor, S. J., Scott, K., Uhl, A., and Weise, D., 2002, "Model-Driven Architecture," *Lecture Notes In Computer Science*, pp. 290-297.

[35] Michalewicz, Z., 1996, *Genetic Algorithms+ Data Structures= Evolution Programs*, Springer, New York.

[36] Mitchell, E. E. L., and Gauthier, J. S., 1976, "Advanced Continuous Simulation Language (ACSL)," *SIMULATION*, **26**(3), pp. 72-78.

[37] 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.

[38] Modelica Association, 2005, Modelica Language Specification, http://www.modelica.org/documents/ModelicaSpec22.pdf. January 12, 2009.

[39] Murthy, K. V. S., and Salzberg, S. L., 1996, *On Growing Better Decision Trees from Data*, Thesis, The Johns Hopkins University.

[40] Nagl, M., 1979, "Graph-Grammatiken, Theorie, Implementierung, Anwendungen," *Vieweg, Braunschweig*.

[41] Nytsch-Geusen, C., 2007, "The Use of UML within the Modelling Process of Modelica-Models," in *International Workshop on Equation-Based Object-Oriented Languages and Tools*, Linköping University Electronic Press, Berlin, Germany.

[42] Object Management Group, 2007, OMG Systems Modeling Language Specification, http://www.omg.org/cgi-bin/doc?ptc/07-09-01. March 15, 2009.

[43] Object Management Group, 2007, Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, http://www.omg.org/docs/ptc/07-07-07.pdf. March 20, 2009.

[44] Pahl, G., Beitz, W., Feldhunen, J., and Grote, K.H., 2007, *Engineering Design: A Systematic Approach*, Springer, London, UK.

[45] 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.

[46] Peak, R., Friedenthal, S., Moore, A., Burkhart, R., Waterbury, S., Bajaj, M., and Kim, I., 2005, "Experiences Using SysML Parametrics to Represent Constrained Object-Based Analysis Templates," *7th NASA-ESA Workshop on Product Data Exchange (PDE),* Atlanta, GA, USA.

[47] 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.

[48] Peak, R. S., and Wilson, M. W., 2001, "Enhancing Engineering Design and Analysis Interoperability Part 2: A High Diversity Example," *First MIT Conference Computational Fluid and Structural Mechanics (CFSM)*, Cambridge, Massachusetts, USA.

[49] 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.

[50] Pop, A., and Akhvlediani, D., and Fritzson, P., 2007, "Towards Unified Systems Modeling with the Modelicaml UML Profile," in *International Workshop on Equation-Based Object-Oriented Languages and Tools*, Linköping University Electronic Press, Berlin, Germany.

[51] Powell, A., Nilsson, M., Naeve, A., and Johnston, P., 2007, DCMI Abstract Model, http://dublincore.org/documents/2007/06/04/abstract-model/. January 21, 2008.

[52] 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.

[53] Rinderle, J. R., 1991, "Grammatical Approaches to Engineering Design, Part II: Melding Configuration and Parametric Design Using Attribute Grammars," *Research in Engineering Design*, **2**(3), pp. 137-146.

[54] Sage, A. P., and Armstrong Jr., J. E., 2000, *Introduction to Systems Engineering*, Wiley and Sons.

[55] Sasajima, M., Kitamura, Y., Ikeda, M., and Mizoguchi, R., 1995, "FBRL: A Function and Behavior Representation Language," *Proc. of IJCAI*, **95**, pp. 1830-1836.

[56] Schmidt, L. C., and Cagan, J., 1997, "GGREADA: A Graph Grammar-Based Machine Design Algorithm," *Research in Engineering Design*, **9**(4), pp. 195-213.

[57] 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.

[58] Schürr, A., 1994, "Specification of Graph Translators with Triple Graph Grammars," in *WG'94 Workshop on Graph-Theoretic Concepts in Computer Science*.

[59] Schürr, A., 2001, "Adding Graph Transformation Concepts to UML's Constraint Language OCL," *Electronic Notes in Theoretical Computer Science*, **44**(4), pp. 93-106.

[60] Simmetrix Inc., 2006, Simulation Application Suite, http://simmetrix.com/products/SimulationApplicationSuite/main.html. Jun 20, 2006.

[61] Simulink (The Mathworks), 2008, Simulink, http://www.mathworks.com/products/simulink/. Feb 1, 2008.

[62] Stahl, T., Voelter, M., and Czarnecki, K., 2006, *Model-Driven Software Development: Technology, Engineering, Management*, John Wiley & Sons.

[63] Starling, A. C., Street, T., and Shea, K., 2005, "A Parallel Grammar for Simulation-Driven Mechanical Design Synthesis," *ASME International design Engineering Technical Conferences (IDETC/CIE2005)*, **2**, pp. 24-28.

[64] Stone, R. B., and Wood, K. L., 2000, "Development of a Functional Basis for Design," *Journal of Mechanical Design*, **122**, pp. 359-370.

[65] SysML, 2006, OMG Systems Modeling Language (OMG SysML), V1.0, http://www.omgsysml.org/. June 20, 2006.

[66] Szykman, S., Sriram, R., Bochenek, C., and Racz, J., 1998, "The Nist Design Repository Project," *Advances in Soft Computing - Engineering Design and Manufacturing*, Springer-Verlag, London, pp.

[67] Tzilla, E., Robert, E. F., and Atef, B., 2001, "Aspect-Oriented Programming: Introduction," *Communications of The ACM*, **44**(10), pp. 29-32.

[68] Ulrich, K., and Tung, K., 1991, "Fundamentals of Product Modularity," *1991 ASME Design Technical Conferences - Conference on Design / Manufacture Integration*, Miami, Florida.

[69] 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.

[70] 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.

[71] Weisemoller, I., and Schurr, A., 2007, "A Comparison of Standard Compliant Ways to Define Domain Specific Languages," *4th International Workshop on Software Language Engineering*, pp. 31-45.