INTEGRATING MODELS AND SIMULATIONS OF CONTINUOUS DYNAMIC SYSTEM BEHAVIOR INTO SYSML

A Thesis
Presented to
The Academic Faculty

by

Thomas A. Johnson

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

Georgia Institute of Technology
August 2008

INTEGRATING MODELS AND SIMULATIONS OF CONTINUOUS DYNAMIC SYSTEM BEHAVIOR INTO SYSML

Approved by:

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

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

Dr. Russell Peak
Manufacturing Research Center
Georgia Institute of Technology

Date Approved: April 19th, 2008

ACKNOWLEDGEMENTS

As my time at Georgia Tech comes to its conclusion, I can't help but reflect upon my journey through life. I realize that I have attained relative success and attribute part of that to my drive for self improvement; however, I know that I would never have gotten to this point in my life without the invaluable help of many different people.

I must first thank my family because they have played the greatest role in my life. My mom, Amy, has always believed in me and ensured that I was on the path to academic success. Through the good times and the bad times, she has been my most important source of support. I also owe gratitude to my dad, Dan, because of his support, encouragement, and pride in my success. I must also thank my other parents, Gary and Sherry. I find it hard to refer to them as "step-parents" because they always treat me like I'm their own flesh and blood. My brother, Andrew, and my cousin, David, also deserve acknowledgement because of their continual help and open ears. Finally, I must thank my grandparents for their encouragement and inspiration. My grandmothers, Jean and Delta, are very proud of me and make sure that I know it. My grandfathers, David and Bill, have been key inspirations in my life and I continually strive to be as successful as them.

Outside of my family, no one has contributed to my success as much as Alice Hoang. Her love and support are uncompromising and for that I am truly grateful.

During my academic career, I've met several inspiring teachers, but none of them have impressed and helped me as much as Dr. Chris Paredis. I am consistently surprised by the depth of his knowledge and inspired by his vision for the future. I have learned

much under his guidance and I can't imagine undertaking an engineer career without that knowledge.

I must thank my academic family in the SRL and especially the folks in MaRC 266 and 267. First of all, I have to thank Rich Malak. He has been a great friend and provided me with an endless amount of guidance. I also have to thank Manas Bajaj for his friendship, insight, and intellectual guidance. Other outstanding individuals include Jonathan Jobe, Alek Kerzhner, Roxanne Moore, and Stephanie Thompson. These people are a perpetual source of knowledge and fun and I thank them for their friendship.

I must also thank Roger Burkhart at Deere & Company, Sanford Friedenthal at Lockheed Martin, and Dr. Leon McGinnis here at Georgia Tech. They have been important contributors to and champions of my work. Gratitude is also due to my reading committee, Dr. Russell Peak and Dr. Dirk Schaefer, for their interest and intellectual contributions.

Finally, I appreciatively acknowledge the support of the George W. Woodruff School of Mechanical Engineering, Deere & Company, and the Center for Compact and Efficient Fluid Power supported by the National Science Foundation under Grant No. EEC-0540834. Additionally, I am grateful for the academic software licenses provided by Embedded Plus Engineering, IBM, Dynasim, and No Magic Inc.

TABLE OF CONTENTS

LIST OF FIGURES

Page

SUMMARY

The objective of this research is to use graph patterns and transformation rules to integrate models of continuous dynamic system behavior with SysML information models representing systems engineering problems. The driver behind this objective is the current state of systems engineering. Contemporary systems engineering problems are becoming increasingly complex as they are handled by geographically distributed design teams, constrained by the objectives of multiple stakeholders, and inundated by large quantities of design information. According to the principles of model-based systems engineering (MBSE), engineers can effectively manage increasing complexity by replacing document-centric design methods with computerized, model-based approaches for representing and investigating their knowledge during system decomposition and definition.

In this thesis, modeling constructs from SysML and Modelica are integrated to improve support for MBSE. The Object Management Group has recently developed the Systems Modeling Language (OMG SysML™). This visual modeling language provides a comprehensive set of diagrams and constructs for modeling many common aspects of systems engineering problems (e.g. system requirements, structures, functions, and behaviors). Complementing these SysML constructs, the Modelica language has emerged as a standard for modeling the continuous dynamics (CD) of systems in terms of hybrid discrete- event and differential algebraic equation systems.

The integration of SysML and Modelica is explored from three different perspectives: the definition of CD models in SysML; the use of graph transformations to automate the transformation of SysML CD models into corresponding Modelica models;

and the integration of CD models and other SysML models. The ability to define CD models is established through a language mapping between SysML and Modelica. The mapping is then used to support model transformations through the creation of a triple graph grammar and corresponding graph transformation rules. Finally, CD models are integrated with other SysML models (e.g. structural, requirements) through the depiction of simulation experiments and engineering analyses. Throughout the thesis, example models of a car suspension and a hydraulically powered excavator are used for demonstration.

The core of this work is the establishment of modeling abilities that do not exist independently in SysML or Modelica, but only as a result of integration. These abilities include enabling systems engineers to model CD in SysML, automatically generate an executable Modelica model from a SysML model, and prescribe necessary system analyses and explicitly relate them to stakeholder concerns or other system aspects. Moreover, this work provides a basis for model integration which can be generalized and re-specialized for integrating other modeling formalisms into SysML.

CHAPTER 1

INTRODUCTION

Our society relies on the everyday operation of engineered systems. From power plants to automobiles to personal computers, engineered systems greatly affect many aspects of our daily lives; however, routine exposure to these systems makes it easy for us to overlook their immense complexity. Contemporary complex systems function at many different physical scales; contain multiple subsystems and components; exhibit emergent behavior that is not readily comprehensible by examining component behavior; encompass multiple engineering disciplines; and are constrained by the objectives of multiple stakeholders. Accordingly, contemporary systems engineering problems involve large quantities of interdependent design information that must be transformed though a systematic design process into a complete system description.

As if systems engineering problems themselves didn't provide enough complexity for engineers to manage, globalization is now adding its own complications. Decades ago, most systems were engineered in one geographical location; however, to maintain a competitive edge in the present global marketplace, businesses must now employ engineering services from the most cost effective and capable sources regardless of location. Consequently, design teams undertaking systems engineering problems are increasingly composed of modular units that operate in multiple geographical locations. Additionally, these design teams consist of a heterogeneous membership of system analysts, component-level disciplinary engineers, and system-level engineers. Communication amongst team members can be hindered by the fact that different disciplines rely on different notations and views of the same system knowledge and

information. Clearly, the coordination of a globally dispersed, multidisciplinary design team coupled with the inherent complexity of a contemporary systems engineering problem imparts a monumental information management problem upon systems engineers.

1.1 Managing Complexity with Model-Based Systems Engineering (MBSE)

As complexity grows in a systems engineering problem, engineers must effectively manage an increasing quantity of intricate design knowledge and information. Accordingly, problems encountered during systems engineering projects are generally correlated with the organization and management of complexity rather than with the direct technological concerns that affect individual subsystems and specific physical science areas [1]. If engineers cannot effectively manage project complexity, they might overlook important design details and dependencies. Such mistakes can compromise stakeholder objectives and lead to costly design iterations or system failures.

Traditionally, systems engineering problems are solved using systematic design processes such as the method prescribed by Pahl and Beitz [2] or the systems engineering "Vee" model proposed by Forsberg and Mooz [3]. Systematic design processes consist of sets of information transformations that iteratively convert stakeholder objectives and requirements into a complete system description. As seen in Figure 1.1, the inputs and outputs of each transformation are generally documents containing the necessary system knowledge and information.

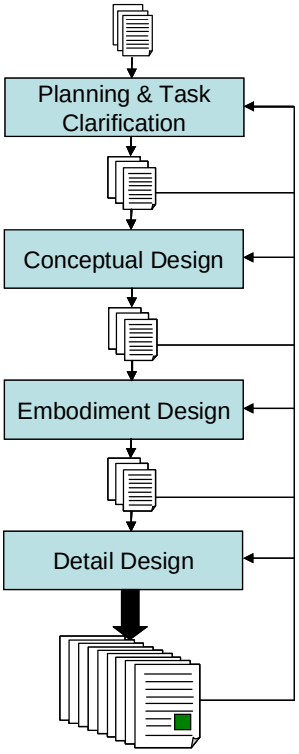

Figure 1.1: Document-centric design.

Furthermore, the final output of the design process is a large collection of product documentation used to support the subsequent lifecycles (e.g. manufacturing, deployment, or disposal) of the system.

While document-centric design coupled with hard work proved to be a successful combination for solving systems engineering problems in the past, it may become inadequate for dealing with the current increases in system complexity and globalization. To transfer knowledge and information between design team members or steps in the design process, engineers must navigate the relevant documents, extract the necessary knowledge/information, and translate that content into discipline-specific (e.g. mechanical, electrical, computer science) formats. This can be a cumbersome and error-prone task. Incorporating the effects of globalization only exacerbates the matter.

Moreover, increasing system complexity correlates with growing quantities of system information; hence, more labor is needed to decipher product documentation.

To cope with increasing complexity and globalization, engineers can adopt model-based design methods for solving systems engineering problems [4]. Model-based systems engineering (MBSE) [5] encourages engineers to move away from document-centric design and towards a more computer-based, interactive modeling approach. Using an MBSE approach to systems design, engineers solve systems engineering problems through the formal elaboration of models that transform stakeholder requirements and objectives into a full system description. In particular, these models are used to describe formally the structure, function, and behavior of a system [6].

The MBSE design approach requires the development of many different design and analysis models. Design models are used to specify the desired structure, function, and behavior of the system. Example design models include models of system architecture, CAD models, and use case models. Analysis models, on the other hand, are used to analyze the anticipated behavior of the system. Example analysis models include models of continuous dynamic system behavior, finite element models, and cost models.

If engineers adopt a MBSE design approach, they are given the valuable capability to share more easily the critical knowledge and information captured in various design and analysis models. Exploiting this capability can thwart problems related to information traceability and consistency that are often encountered in document-centric design processes. Consequently, engineers must integrate the critical knowledge captured in design and analysis models. Ideally, integration could be achieved through the sole use of one modeling language that is able to depict all aspects of a systems

engineering project at every necessary level of fidelity; however, the creation of such a modeling language is not a realistic endeavor. Moreover, such a language would simply reinvent the abilities of other domain-specific modeling languages.

Alternatively, to achieve model integration the knowledge needed to make design decisions should be abstracted from domain-specific models into a system information model. An information model as described by Mylopoulos [7] is a computer-based symbol structure that formally captures and organizes information in a meaningful fashion. The information model then serves as a platform for model integration and only exposes knowledge and information that is important to the design team as a whole. The unnecessary details remain encapsulated in smaller design or analysis models for individual use.

While model integration is an import function of an information model, it also serves other purposes. The information transformations occurring in a MBSE design process, in contrast with traditional methods, are recorded in the information model rather than in large sets of documentation. Furthermore, the primary output of an MBSE design process is the information model which is subsequently used to support the later lifecycles of the system.

The MBSE approach to systems design, as depicted in Figure 1.2, offers some important benefits for engineers coping with complex systems and globally distributed design teams.

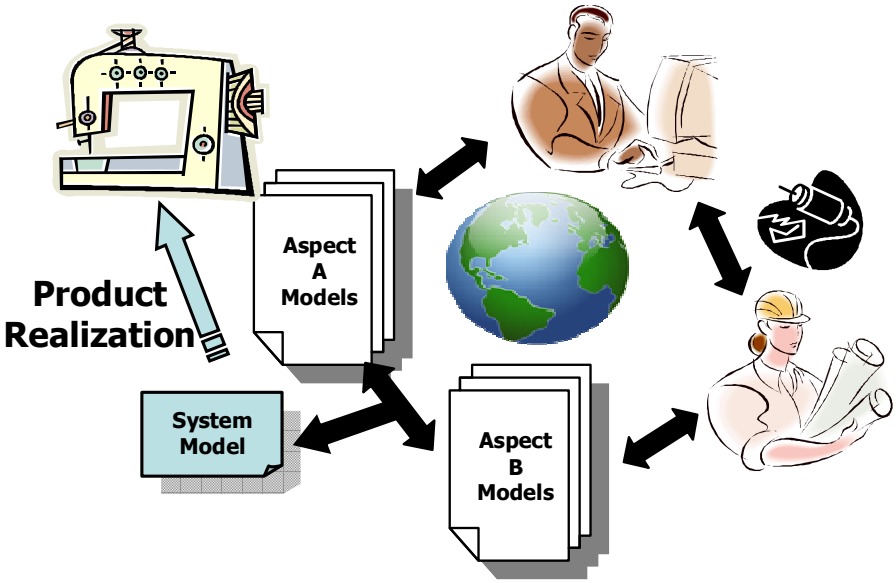

Figure 1.2: A globally distributed, MBSE approach to systems design.

The information generated in the design process is stored in one central location (e.g. a computer server) that is accessible by any member of the design team regardless of geographical location. This promotes close collaboration amongst designers who have no physical contact with each other. Assuming that the information model is authored using a well-understood modeling language, the team members also have a strict protocol for communicating important design knowledge and information. Additionally, all the contents of an information model generally exist in one modelspace, but can be displayed to different individuals in various fashions using multiple views or diagrams. This is analogous to displaying the same system information in different documents for different design team members; however, multiple documents permit the existence of information consistencies. This is not the case when using multiple views of the same information model.

1.2 Using SysML in Support of MBSE

Several information modeling formalisms have been developed in support of MBSE design processes. Two well-known information modeling languages are the Object Management Group's (OMG) successful Unified Modeling Language (UML™) [8] and the recently adopted Systems Modeling Language (OMG SysML™) [9].

UML is a graphical modeling language for specifying, constructing, and documenting the artifacts of software, business models, and other applicable systems. It is a general-purpose modeling language that can be used with all major object and component methods. The language is commonly used during the development of large-scale, complex software for various domains and implementation platforms [10].

SysML is also a general-purpose systems modeling language that enables engineers to create and manage information models of engineered systems using well-defined, visual constructs [9]. Instead of developing SysML as an original language, the OMG extended UML for the systems engineering community. SysML reuses and extends a subset of UML 2.1 constructs:

- it extends UML classes with *blocks*;
- it supports requirements modeling;
- it supports parametric modeling;
- it extends UML dependencies with *allocations*;
- it reuses and modifies UML *activities*;
- it extends UML standard ports with *flow ports*.

Figure 1.3 depicts the SysML diagram taxonomy as a graphical representation of SysML's extension of UML.

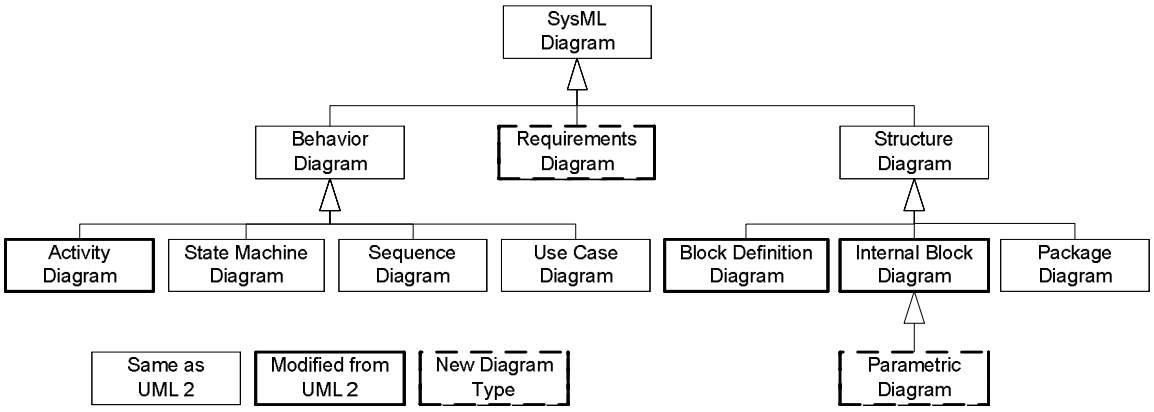

Figure 1.3: The SysML diagram taxonomy [9].

A block with a regular or bold border represents a UML diagram that has been reused or modified, respectively. Blocks with a dashed border represent new diagrams, namely, the *requirements* and *parametric diagrams*.

The knowledge captured in a SysML model is intended to support the specification, analysis, design, and verification and validation of any engineered system [9]. As a result, SysML is commonly used to model system requirements, tests, structures, functions, behaviors, and their interrelationships. While capturing all of the above knowledge is critical for ensuring success in solving a systems engineering problem, modeling system behavior is arguable most important. If a system does not behave in a way that satisfies stakeholder objectives, then it is useless regardless of its other aspects.

1.3 Modeling System Behavior with SysML

SysML is capable of depicting system behavior using the following language constructs:

- *Activity diagrams* describe the inputs, outputs, sequences, and conditions for coordinating various system behaviors;

- *Sequence diagrams* describe the flow of control between actors and a system or its components;
- *State machine diagrams* are used for modeling discrete behavior through finite state transition systems;
- *Parametric diagrams* allow users to represent mathematical constraints amongst system properties.

The first three of these modeling constructs support causal behavioral modeling in terms of discrete events. The last one enables a user to model equations (called *constraints* in SysML) that establish mathematical relationships between the properties of a system or its components. While SysML offers many behavioral modeling *capabilities* with the above constructs, the language specification does not explicitly provide the *ability* to integrate many different types of behavioral models required to solve systems engineering problems.

Oftentimes, engineers need to analyze the continuous dynamics (CD) of a system alternative. CD are generally represented by hybrid discrete event and differential-algebraic equation (DAE) models which characterize the exchange of energy, signals, or other continuous interactions between system components; however, the SysML specification provides no explicit support for integrating DAE models into SysML models. In other words, no guidance is provided for integrating models authored in languages like Modelica [11] or Matlab/Simulink [12]. The intent of the work presented in this thesis is to overcome this burden by building upon SysML's current capabilities.

1.4 Motivating Questions

As depicted in Figure 1.4 and discussed in Section 1.1, SysML is not simply an information modeling language, but is really a platform for model integration.

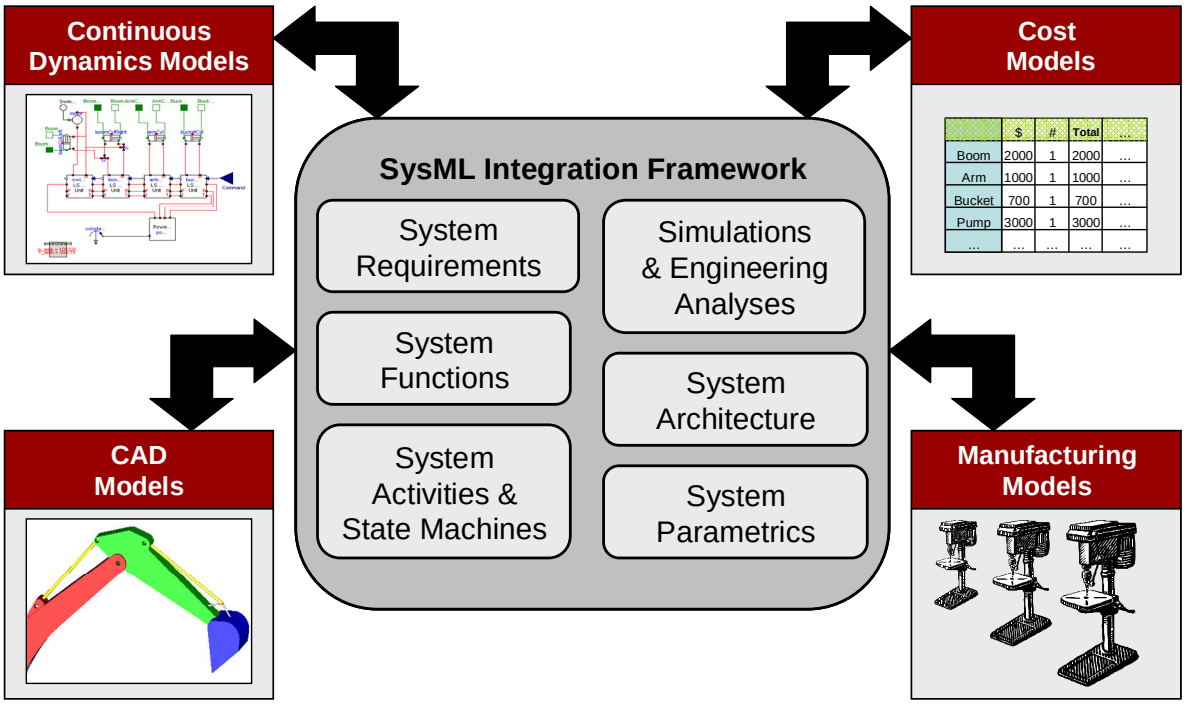

Figure 1.4: SysML as a model integration platform.

Using SysML constructs independently of outside languages or tools, modelers can author several different types of systems engineering models in SysML (e.g. requirements models, use case models, activity models). While these types of models are certainly *necessary*, they are not *sufficient* for ensuring the success of SysML. To improve SysML's ability to support MBSE design processes, the following question must be answered:

The Motivating Question:

How can engineers integrate models in various formalisms with SysML information models to promote information consistency, model traceability, and automated model transformation?

As stated in the question, solutions for model integration should improve support of MBSE design processes through the following benefits: information consistency, model traceability, and automated generation of executable models from SysML models. By integrating the important knowledge and information contained in various engineering models used to solve systems engineering problems, engineers can ensure information consistency throughout the various models used to solve a systems engineering problem. Additionally, integration enables the tracing of important associations and dependencies amongst the various models. Lastly, information consistency and traceability can enable engineers to set the context for system analyses that encompass multiple engineering models. This enables the automated population of consistent information into executable models used to analyze a system.

While the question of model integration is the central motivation for this thesis, it is too broad to be answered in full. Instead, this thesis limits the scope of the question to integrating CD models into SysML. To answer this reduced motivating question, it is decomposed into three manageable sub-questions. The first question investigates the actual SysML depiction of CD models built upon sets of DAEs:

Question 1:

How can engineers effectively represent models of continuous dynamic system behavior using the modeling constructs offered in SysML?

The answer to this question is the foundation for CD model integration. If external CD models can be appropriately abstracted or represented using SysML modeling constructs, then CD model integration and the resultant information consistency and traceability can come to realization.

The representation of CD models using SysML modeling constructs is only the first step to integrating CD models into SysML. True integration can only come to fruition when a SysML CD model can be linked to an external, executable CD model. Such a linkage can be accomplished through model or graph transformations. Graph transformations enable the automated, external execution of a non-executable SysML CD model and the integration of an external CD model into a SysML system information model. Additionally, it provides a method for ensuring information consistency between an external CD model and a SysML CD model.

In this thesis, Modelica [11] is the external CD modeling language of interest. 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 [13] or ACSL [14]. In addition, both SysML and Modelica are similar in that they use base modeling elements that adhere to the principles of object-oriented modeling.

Since Modelica is the CD modeling language to be integrated with SysML, the following question is posed:

Question 2:

Are graph transformations an effective means of transforming between SysML models of continuous dynamic system behavior and corresponding Modelica models to enable automated model execution and to ensure information consistency?

The answer to this question is the key to automating the integration of SysML and Modelica models. An explicit model transformation schema can be incorporated in a computer program used to transform from SysML to Modelica or vice-versa.

The answers to Questions 1 and 2 enable the integration of SysML and Modelica models, but don't explicitly provide guidance on maintaining information consistency and traceability between integrated CD models and other aspects of a SysML information model. During the course of a systems engineering problem, many different models (e.g. structural models, CD models, objective function models, requirements models) are used to make decisions concerning a system alternative's fulfillment of stakeholder requirements and objectives; hence, a decision maker must fully understand the relationships between these models. To ensure that a decision maker understands these relationships, explicit traceability can be established between the necessary models. With respect to a system's continuous dynamic behavior, the relationships between CD models and other models exist in the context of a system simulation or analysis. This leads to the following question:

Question 3:

Can engineers ensure model traceability between CD models and other SysML models by explicitly modeling simulations and analyses of system alternatives in SysML?

A promising answer to this question is essential for using SysML as an integration platform in support of decision making. If a SysML representation of a CD model can't be related to other SysML models in a meaningful fashion, then its inclusion in an information model provides little value.

1.5 Thesis Overview

According to the motivating questions in Section 1.4, the objective of the work presented in this thesis is to use graph patterns and transformation rules to integrate models of continuous dynamic system behavior with SysML information models representing systems engineering problems. This is depicted graphically in Figure 1.5.

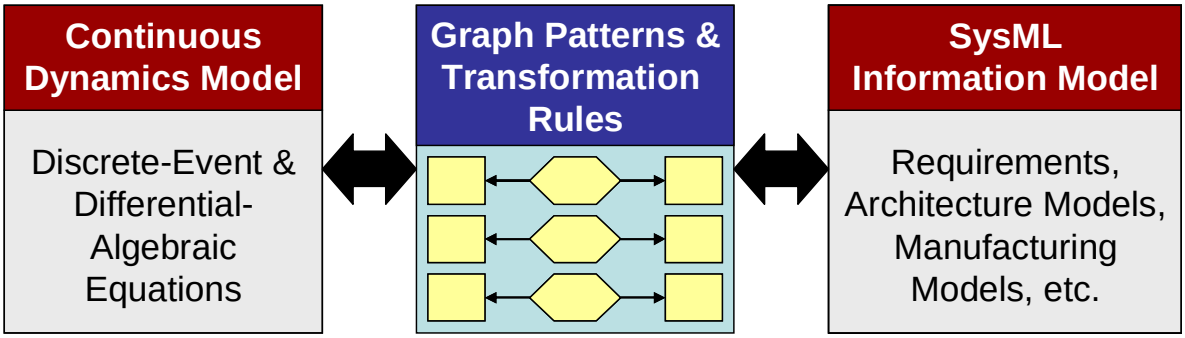

Figure 1.5: The research objective.

By achieving this objective, the vision for model integration as depicted in Figure 1.4 can take one more step towards reality. Disciplinary or component-level designers can use external languages and software tools for creating detailed, low-level design and analysis

models. Then, they can abstract into the SysML system information model the knowledge and information that is relevant at the system level. Once the information/knowledge is abstracted into a SysML model, it can be bound or associated with other models (e.g. models of simulations, engineering analyses, requirements, system structure, use cases) created in or abstracted into a SysML system information model. Using such relationships, the elements of an integrated model can be updated via the SysML system information model and reflected in the external design and analysis models through the use of automated model/graph transformations. If the modelers use such tools for transforming from SysML to external languages/tools and vice-versa, true model integration can become a realization via the bidirectional flow of information/knowledge.

Before acceptable answers can be provided for the motivating questions, we must have a better understanding of the extent to which they have already been answered. Accordingly, Chapter 2 of this thesis provides an overview of work that is highly related to the motivating questions. Due to the strong tie between this thesis and SysML, Section 2.1 provides a review of some important SysML constructs and introduces the car suspension example used in later chapters. This section is specifically aimed at readers who have limited or no familiarity with SysML and UML. Readers who are familiar with both languages need not delve into the details. Section 2.2 provides an overview of work concerning the integration in SysML of design and analysis models. Section 2.3 then provides a more specific overview of relevant attempts at integrating CD models into SysML. Finally, Section 2.4 highlights relevant work in the field of model transformations.

To answer Question 1, Chapter 3 describes in detail the approach to representing models of continuous dynamic system behavior in SysML. This is accomplished through the specification of a modeling approach and set of SysML constructs that correspond to important Modelica modeling practices and constructs. When a clear mapping between the two languages does not exist, a SysML extension is provided to fill the gap.

To answer Question 2, Chapter 4 explains an approach to transforming SysML models into Modelica models and vice-versa. The approach relies on a triple graph grammar (TGG) [15] and a corresponding set of graph transformation rules. The automated transformation process is implemented using the VIATRA [16, 17] model transformation framework and Eclipse [18]/Rational Systems Developer (RSD) [19]..

To answer Question 3, Chapter 5 provides an approach and set of SysML constructs for supporting decision-making processes through the explicit SysML depiction of CD simulations and engineering analyses. The approach is broken down into four steps: establishing the context of a CD model with respect to a system alternative, modeling the simulation, abstracting the simulation into an input-output model, and embedding the simulation in an engineering analysis.

The final three chapters bring this thesis to a close. To demonstrate several important concepts described in this thesis, Chapter 6 exhibits the SysML integration of a CD model of hydraulically powered excavator. Chapter 7 is then intended to discuss, evaluate, and draw some important conclusions about the work described in this thesis.

CHAPTER 2

RELATED WORK

2.1 An Introduction to SysML

Before discussing any relevant work or the approach for integrating CD and simulation models in SysML, this section reviews some important SysML constructs and introduces an example problem used throughout this thesis.

2.1.1 SysML Blocks

The primary modeling unit in SysML is the *block*. As described in Chapter 8 of the SysML specification [9], 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.

All block declarations occur in a *Block Definition Diagram* (BDD). A BDD is used to define block features and the relationships between blocks or other SysML modeling elements. Figure 2.1 is a BDD depicting the definition of a car and its suspension. A car is obviously composed of more subsystems and components, but Figure 2.1 is sufficient for the sake of demonstration. SysML allows a modeler to omit elements of the underlying information model that detract from the main intent of a diagram.

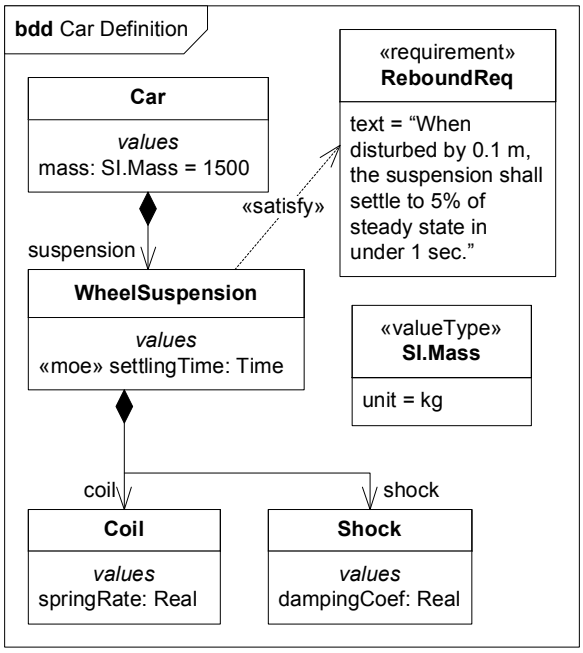

Figure 2.1: A SysML model of a car and its suspension.

2.1.2 SysML Value Types

A SysML *value type* is an extension of the UML *data type* used to define types of values that may be used to express information about a system [9]. More specifically, value types are used to assign to a value property the units and dimensions declared in its definition. For example, Figure 2.1 displays the definition of *SI.Mass* which carries units of kilograms.

2.1.3 SysML Properties

A SysML property describes a part or characteristic of a block and consists of a named value of a specified type. In Figure 2.1, two important types of properties are depicted. The first kind is the *part property*. Part properties represent a subsystem or component of a system and must be typed by a block. Part properties can be depicted in the *parts compartment* of a block or using a *composition association*. A composition

association is displayed as a black diamond with a tail. The property name appears at the tail end of the association. For example, the block *Car* in Figure 2.1 owns a part property named *suspension* of type *WheelSuspension*.

The second kind of property is a *value property*. A value property appears in a block's *values compartment* and represents a quantifiable characteristic of a block (e.g. mass, length, velocity). Accordingly, it must be typed to a SysML value type or UML data type. For example, *Car* in Figure 2.1 has a value property *mass* which is typed to the value type *SI.Mass* to supply units of kilograms.

2.1.4 UML Stereotypes

A *stereotype* is a UML construct used to create customized classifications of modeling elements. Stereotypes are defined by keywords that appear inside of guillemets. These customization constructs extend the standard elements to identify more specialized cases important to specific classes of applications. Most SysML constructs have been defined as UML stereotypes and users are allowed to create additional stereotypes to capture the specialized semantics of a particular application domain. An example of a stereotype is illustrated in Figure 2.1. The stereotype *«moe»* applied to the *WheelSuspension*'s value property *settlingTime* indicates that it is a "measure of effectiveness".

2.1.5 SysML Constraint Blocks

As defined in the SysML specification [9], a *constraint block* is a specialized form of the SysML block and is intended to package commonly used constraints in a reusable, parameterized fashion. Constraint blocks can be identified by the *«constraint»* stereotype

that appears in their namespace compartments. The properties of constraint block are referred to as *parameters* to emphasize the objective of constraint parameterization.

2.1.6 SysML Requirements

A SysML *requirement* is used to represent a textual requirement or objective for a system, subsystem, or component. Requirements are shown with the *«requirement»* stereotype and optionally display a compartment for displaying text and identification fields. Requirements are related to other modeling elements using various *dependencies* such as the *satisfy* and *verify* dependencies. A dependency is a UML construct for expressing different types of relationships between various modeling constructs. The use of SysML requirements and dependencies is demonstrated in Figure 2.1 by the satisfy dependency between *WheelSuspension* and the *ReboundReq* requirement.

2.2 Integrating Design and Analysis Models in SysML

> "Currently it is common practice for systems engineers to use a wide range of modeling languages, tools and techniques on large systems projects. In a manner similar to how UML unified the modeling languages used in the software industry, SysML is intended to unify the diverse modeling languages currently used by systems engineers." [9]

This excerpt from the SysML specification clearly indicates that the intent of the language is to provide a platform for model unification (i.e. integration). The constructs provided by the language are certainly capable of supporting model integration, but they don't necessarily endow a SysML user with the "out of the box" ability to perform model

integration. Rather than relying on end users to enable model integration, this ability should be cultivated by knowledgeable SysML champions.

One notable means of enabling model integration in SysML has been provided through the development of Composable Objects (COBs) [20-22]. 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. COBs recently served as the basis for the development of the SysML parametric diagrams [9]. By establishing a mapping between COBs and SysML parametrics, the integration and execution of engineering analyses (such as structural finite element analyses) within the context of SysML has been demonstrated [23]. This thesis extends the work on COBs by focusing on the integration of CD Modelica models into SysML.

Huang et al. [24] explore the model integration capabilities of SysML through the SysML representation of design and simulation (i.e. analysis) models for manufacturing processes. In particular, the authors present the creation of a flow shop model and subsequently map it to a queuing analysis model. Additionally, the authors describe an approach to automating the generation of an executable eM-Plant [25] flow shop model via XPath [26]. This executable model is then used to simulate the SysML simulation model.

The ability to integrate heterogeneous models in SysML has also been demonstrated through the development of Multi-Aspect Component Models (MAsCoMs) [27]. The MAsCoM framework is intended to support model reuse through the establishment of relationships between design models of system components, corresponding analysis models, and the many aspects of a model that pertain to analysis

objectives, stakeholder perspectives, and other elements of MBSE. Within the framework, analysis models are integrated with component models and aspect models such that their semantics of intended use are captured and represented for reuse.

2.3 Integrating CD models into SysML

Recently, several researchers have also recognized the need to integrate models of continuous dynamic system behavior into SysML. The approaches to integrating CD models are as varied as the CD modeling languages being integrated. In this section, several approaches are reviewed and contrasted with the approach outlined in Chapter 3 of this thesis.

Currently, Matlab/Simulink models of system dynamics are used extensively in the development of engineered systems. Recognizing this dependency, Vanderperren and Dehaene [28] have discussed the current and future states of UML/SysML and Matlab/Simulink integration using two different approaches: co-simulation and reliance upon a common execution language. The intent of both approaches is to test the design of an embedded system and its control software by simultaneously executing a UML model of the software and a Simulink model of the system dynamics. The co-simulation approach involves data exchange between a UML tool and Simulink via an interface tool. This approach is demonstrated by Hooman et al. [29] and implemented in Telelogic's Rhapsody [30] UML modeling tool. The other approach, demonstrated in the GeneralStore integration platform [31], relies on the generation and coupling of executable code (e.g. C/C++ code) from both the UML and Matlab/Simulink models. The work presented in this thesis is very similar to these Matlab/Simulink and UML/SysML integration efforts, but adopts the perspective that an information model

should serve as an integration platform rather than as a means for describing only certain aspects of the system.

Another common formalism for modeling continuous dynamic system behavior is the bond graph. Developed in 1961 by Paynter [32], bond graphs are graphical models used to describe continuous dynamics resulting from energy flow through a system and its composition of discrete components. Due to the prevalence and history of bond graphs in systems engineering analysis, Turki and Soriano [33] extended the capabilities of SysML activity modeling to support the representation of bond graphs. While this extension enables bond graph modelers to integrate their models into larger SysML models, the authors only discuss the possibility of generating executable CD models and do not provide guidance for relating SysML bond graph models to other SysML models.

Two groups have worked on the integration of Modelica CD models into SysML/UML. The first work from Fritzson, Akhvlediani, and Pop [34] provides support for modeling continuous dynamics in SysML via the ModelicaML profile for UML/SysML. The ModelicaML profile enables users to depict a Modelica CD model graphically alongside other aspects of a 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.

The second work is a similar profile named UML^{H}. This profile was created by Nytsch-Geusen [35] for developing and graphically depicting hybrid discrete and DAE models in UML/SysML. The author presents hybrid models as Modelica models that are based on a combination of DAEs and discrete state transitions modeled with the

Modelica state chart 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 thesis, the capabilities of ModelicaML and UML^{H} are further extended by demonstrating the integration of CD models with other SysML constructs for requirements, structure, and design objectives. Additionally, this thesis demonstrates the use of model transformations to enable the automated transformation of information between SysML and Modelica models.

2.4 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 [36].

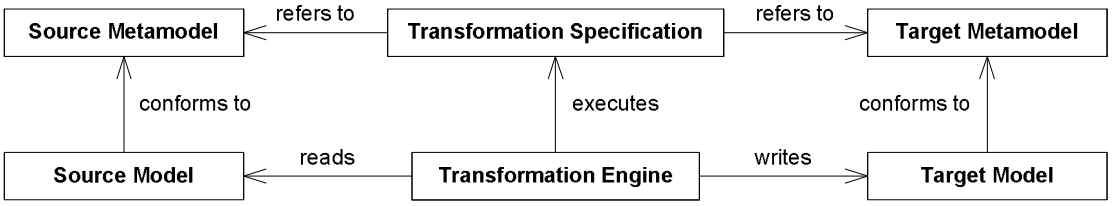

Figure 2.2: The basics concept of model transformation [36].

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. The work presented in this thesis (see Chapter 4) demonstrates the potential of model transformations for MBSE through the generation of executable, lower-level Modelica code from higher-level SysML CD 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) [37] and TGGs [15].

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.

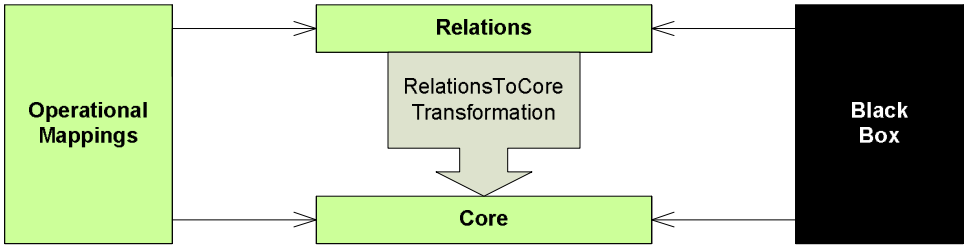

Figure 2.3: Relations between the QVT languages [37].

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 [38]. 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. By querying a model space containing SysML or Modelica models, transformations rules are executed until the model space complies with the specified TGG. For example, Figure 2.4 displays a small TGG that relates a SysML block to a Modelica class using a correspondence entity named *block2class* with one relation pointing to the *block* entity (in the SysML metamodel graph) and one to the *class* entity (in the Modelica metamodel graph).

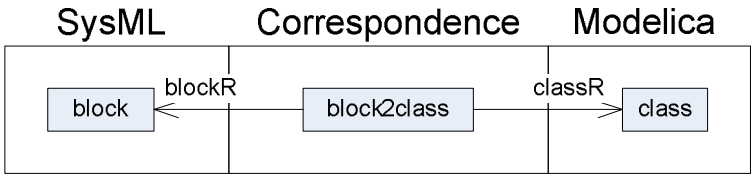

Figure 2.4: An example TGG.

A practical implementation of TGGs is also demonstrated extensively by Königs [39].

2.5 Summary

This chapter provides an overview of material that is highly relevant to model integration via SysML. Section 2.1 is a general introduction to SysML and establishes the context for the example SysML car model used throughout the rest of this thesis. Section 2.2 is a review of some past and ongoing work on various types of model integration via SysML. Section 2.3 is a more specific review of work regarding the integration of CD models into SysML. Section 2.4 is a review of work related to the automation of model synchronization and generation via model/graph transformations.

The work presented in this thesis is clearly part of a larger, ongoing effort to improve MBSE through model integration. It builds upon past and current work in an effort to increase the modeling capabilities of engineers designing complex systems.

This is accomplished by enabling the definition, automated transformation, and integration of CD models into SysML. Moreover, generalizing the work presented in this thesis provides a stencil for integrating other types of design or analysis models into SysML via language mappings, graph transformations, and the depiction of simulations and engineering analyses.

CHAPTER 3

MODELING CONTINUOUS DYNAMIC SYSTEM BEHAVIOR IN SYSML

In this chapter, an approach is described for representing CD models using SysML modeling constructs. More specifically, the approach enables the integration of Modelica-based CD models. First, an approach is outlined for creating fully detailed "white box" CD models in SysML. Then, an approach is outlined for creating low fidelity "black box" CD models in SysML that act as references to existing, external Modelica models.

3.1 Objectives

A model is only valuable if it increases a decision maker's ability to design a better system at an acceptable cost [40] The model for representing CD models in SysML is valuable if it strikes an appropriate balance between the benefits expected from developing a model and the costs of encoding the required information. To develop a valuable modeling approach, the following objectives are established:

1. The approach must enable the integration of continuous dynamics models into broader SysML models. By integrating a Modelica-based CD model into SysML, decision makers can formally recognize relationships between continuous dynamic behavior and other aspects of the system.
2. The approach must facilitate the transformation of SysML CD models into Modelica models and vice-versa. SysML is a language for describing information and knowledge in the context of systems engineering, but is by itself not an

executable language—model execution is relegated to simulation tools. Hence, seamless connections should be established between SysML and CD simulation tools via SysML-to-Modelica model transformations.

3. The approach must encourage model reuse. If a designer can avoid creating every model from scratch by reusing or modifying pre-existing models, he or she can realize significant reductions in the use of project resources.
4. The approach must facilitate efficient stakeholder communication. Unambiguous communication is very important during the development of a complex system. By relying on a formal, accepted approach for integrating CD models in SysML information models, behavioral knowledge can be unambiguously shared amongst designers or stakeholders.

3.2 Modelica as the Foundation

In this thesis, Modelica is the foundation for integrating CD models into SysML. As discussed in Section 1.4, Modelica has emerged as a language of choice for modeling continuous dynamic system behavior. 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 between the languages such that a one-to-one mapping is not possible. Since SysML is intended to be a general-purpose modeling language, some of the specialized semantics of Modelica do not have direct SysML equivalents. To overcome these differences, the approach has been to find an appropriate balance between converting some implicit Modelica semantics into explicit

constraints in SysML or, when that is not possible/valuable, extending SysML constructs through UML stereotypes.

3.3 Integrating "White Box" CD Models into SysML

Through the mapping of essential Modelica modeling constructs to their SysML counterparts, this section provides an approach to creating "white box" CD models in SysML. This enables modelers to capture nearly every detail of a CD model using native SysML constructs. Accordingly, modelers can create strictly "white box" SysML CD models or hybrid "white/black box" system models (Section 3.4.3)

3.3.1 Model Declaration

The fundamental similarity between SysML and Modelica is the use of objects. The primary modeling unit in Modelica is the *class*. Classes serve as definition templates for modeling the components of other classes [41]. To make Modelica easier to read and maintain, special restricted classes were developed for defining the intended function of a class [11]. Example restrictions are *models*, *connectors*, *types*, and *functions*. While the restrictions are useful, they are not necessary in most cases. One can usually maintain model validity by replacing a restricted class with a regular class; however, exceptions to this heuristic (the Modelica *connector* and *type*) are addressed later in this chapter.

The declaration of a Modelica class maps directly to that of a SysML block. This mapping is established because both the class and the block serve as the base modeling unit in their respective language while sharing similar structures. Blocks, like classes, provide the structure for other objects by acting as block definition templates.

Figure 3.1 is an engineering schematic of a Mass-Spring-Damper (MSD) system. The system is composed of a spring and damper mounted in parallel between two system nodes. A mass and a steady-state detection sensor are connected to the top node while the bottom node is connected to the ground.

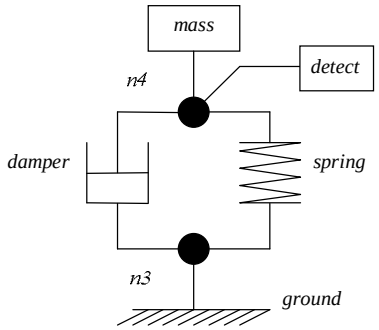

Figure 3.1: An engineering schematic of a MSD system.

Figure 3.2 and Figure 3.3 illustrate the equivalence of SysML blocks and Modelica classes through their representations of CD models corresponding to the schematic in Figure 3.1.

```
//The MSD declaration
class MSD
  //The system components
  Mass mass;
  Spring spring;
  Damper damper;
  Fixed ground;
  SteadyStateDetector detect;
  …
end MSD;

//The Mass declaration
class Mass
  //The variables
  SI.Position s;
  SI.Mass m;
  SI.Velocity v;
  SI.Acceleration a;
  //The interface component
  MechJunction j;
initial equation
  s = -0.1;
equation
  s = j.s;
  v = der(s);
  a = der(v);
  m*a = j.f;
end Mass;

//The MechJunction declaration
connector MechJunction
  SI.Position s;
  SI.Force f;
end MechJunction;
```

Figure 3.2: The declaration of a Modelica representation of a MSD system.

Figure 3.2 is a lexical Modelica model of a Mass-Spring-Damper (MSD). Figure 3.3 displays the corresponding SysML declaration of the MSD CD model.

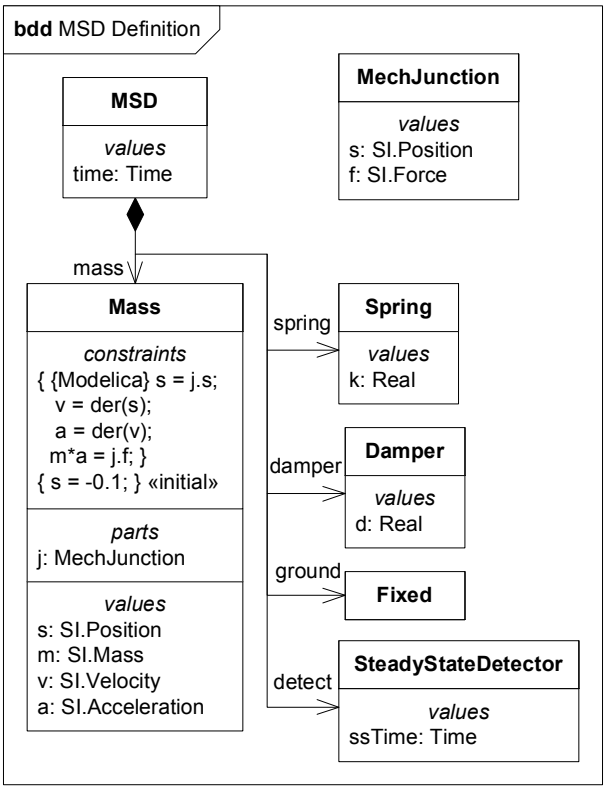

Figure 3.3: The declaration of a SysML representation of a MSD system.

The block *MSD* represents the declaration of the overall MSD system while the other blocks (*Mass*, *Spring*, *Damper*, *SteadyStateDetector*, *Fixed*, and *MechJunction*) represent the definitions of the system components.

In Modelica, the properties of a model are called *components.* A component can represent a part (e.g. spring, damper) or characteristic (e.g. length, position) of the system. One can tell whether a component represents a part or a characteristic by identifying the class to which the component is typed. "Part" components are usages of regular classes or models. These components map to SysML part properties typed to other blocks. "Characteristic" components (i.e. variables) are usages of classes with the *type* restriction. These components and type classes map directly to SysML value properties typed to value types since both are used assign the units of measure or dimension declared in its definition.

The property-component mapping is illustrated in Figure 3.2 and Figure 3.3. For example, in Figure 3.2 the class *MSD* owns a "part" component *mass* typed to the class *Mass*. The class *Mass* owns a "characteristic" component *s* typed to the Modelica type *SI.Position*. This is reproduced in Figure 3.3 by a block *MSD* that owns a part property *mass* typed to the block *Mass*. The block mass owns a value property *s* typed to the value type *SI.Position*.

3.3.2 Model Interface

To interact with other models in an object-oriented (OO) fashion, a given model should have a well-defined interface. Models used in the description of a system's continuous dynamic behavior generally interact using *across* and *through* variables [32] exposed to the rest of the system model. Since across and through variables are the only means of interaction, they can be encapsulated inside of interface objects that are exposed to other system components and subsystems

In Modelica, a model's interface consists of components typed to *connectors*. Modelica connectors are restricted classes that hold across and through variables, but have no equations defining behavior. In Section 3.3.1, Modelica classes were mapped to SysML blocks, so Modelica connectors can also map to blocks. Consequently, a SysML model's interface can be established by creating one or more part properties typed to blocks encapsulating only across and through variables.

To illustrate the declaration of a model interface, Figure 3.3 depicts a block named *MechJunction*. This is a reusable block that encapsulates position and force value properties corresponding to translational across and through variables, respectively. To define the interfaces for each component of *MSD*, the appropriate number of part

properties are declared for each component and then typed to *MechJunction*. For example, *Mass* has one part property *j* typed to *MechJunction*.

3.3.3 Abstraction, Inheritance, and Redefinition

Both languages support model reuse through the OO concepts of abstract classes, inheritance, and redefinition. In this section, a mapping is defined between the SysML and Modelica interpretations of these OO principles.

The first OO principle is the concept of an abstract or partial object. If a Modelica class is tagged with the *partial* keyword, then the class is not fully defined and cannot be instantiated, but serves as a template that can be extended through object inheritance. Similarly, SysML supports the concept of an *abstract* block that exists as a partially defined model.

The second OO principle is object inheritance. Inheritance is a modeling mechanism that enables a child object to inherit and refine the definition of a parent. In Modelica, inheritance is accomplished through the *extends clause*. When inserted in the definition of a Modelica class, the extends clause automatically imports the entire definition of the target (parent) class. Similarly, SysML blocks (and other modeling elements) can be extended through the use of specialization/generalization relationships. A generalization is depicted by an arrow with a white head.

Figure 3.4 illustrates the concepts of a partial class and class inheritance in Modelica.

```
//The partial MechSensor declaration
partial class MechSensor
  MechJunction j;
  …
end MechSensor;

//The SteadyStateDetector declaration
class SteadyStateDetector
  extends MechSensor;
  …
end MSD;
```

Figure 3.4: Demonstration of Modelica OO modeling constructs.

As seen in the figure, the class *SteadyStateDetector* extends the partial class *MechSensor*. This indicates that a *SteadyStateDetector* is a subtype of a *MechSensor* and inherits a component typed to *MechJunction*. The equivalent SysML modeling constructs can be seen in Figure 3.5.

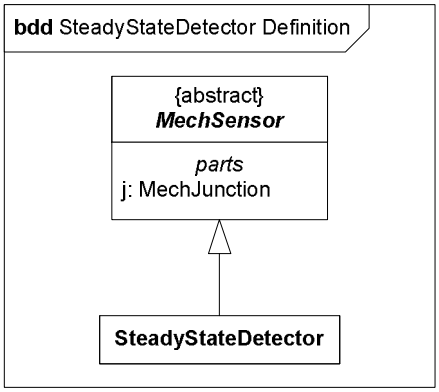

Figure 3.5: Corresponding demonstration of SysML OO modeling constructs.

The SysML block *MechSensor* is partially defined due to *{abstract}* appearing in the block's namespace. *SteadyStateDetector* extends this partial definition through the specialization relationship.

Modelica also supports model reuse through the use of replaceable properties and their redeclaration. A Modelica class can have components that are tagged by the *replaceable* keyword. This allows the component to be redefined using the *redeclare* construct when its owning class is typed by a component in another class. In SysML,

every property of a block is considered to be replaceable using standard UML mechanisms of redefinition.

3.3.4 DAE-Based Internal Behavior

DAEs are commonly used to define the continuous dynamic behavior of a system. To define the DAE-based internal behavior of a class, Modelica employs the *equation clause* in which equations can be used to maintain mathematical relationships between the class's components. Similarly, the creation of mathematical relationships between SysML properties is accomplished by assigning constraints to a given block. Constraints appear between braces and are displayed in a block's *constraints compartment*.

Oftentimes, initial conditions must be placed on a model to ensure that a mathematical solver can provide an analytical or numerical solution to a system of differential equations. In the context of a numerical solution, initial conditions are held true at the beginning of a simulation and can change thereafter. The creation of initial conditions is generally accomplished in Modelica using the *initial equation clause*. To map this concept into SysML, a distinction must be made between regular and initial constraints. Such distinctions or semantic extensions are accomplished in SysML using UML stereotypes. Accordingly, a constraint can be characterized as an initial condition using the *«initial»* stereotype. This stereotype is an original extension to SysML and can only be assigned to constraints. The stereotype specifies that the constraint must be true at the beginning of a simulation.

To illustrate the use of Modelica equations, Figure 3.2 displays the class *Mass* and its behavior as characterized by the initial equation and equation clauses. Equivalent usages of SysML constraints and the *«initial»* stereotype are displayed in Figure 3.3. The

internal behavior of the block *Mass* is defined using four regular constraints and one initial constraint. Note that the constraints explicitly refer to the Modelica language, but other syntax could be used according to the modeler's preferred executable language.

3.3.5 Composing the System Model

Composing a system CD model comprises the description of energy and signal interactions between system components. Generally, such component interactions are modeled using the equivalent of Kirchhoff's circuit laws: at a connection (i.e. system node) all across variables are equal and all through variables add up to zero.

In Modelica, interactions between system components are modeled using Modelica connectors, the *flow prefix*, and *connect clauses.* As discussed in Section 3.3.2, connectors are used to encapsulate across and through variables. Other classes then use these connector definitions to create interface components. The Modelica language offers a unique modeling construct called the *flow prefix* that can be used to explicitly identify a connector's through variables. This is important when composing a system model with Modelica *connect clauses*. A connect clause is a special equation used in a system model's equation section for connecting the interface components of the system components. If two or more connector components are connected with connect clauses, the following equations are implicitly defined: all flow variables sum to zero while any other variables are equal. This is advantageous for modelers because they don't need to model system nodes—the circuit equations (i.e. the equivalent of Kirchhoff's laws) implicitly exist in the model. The lack of explicit system nodes is illustrated in the Modelica connection diagram of Figure 3.6.

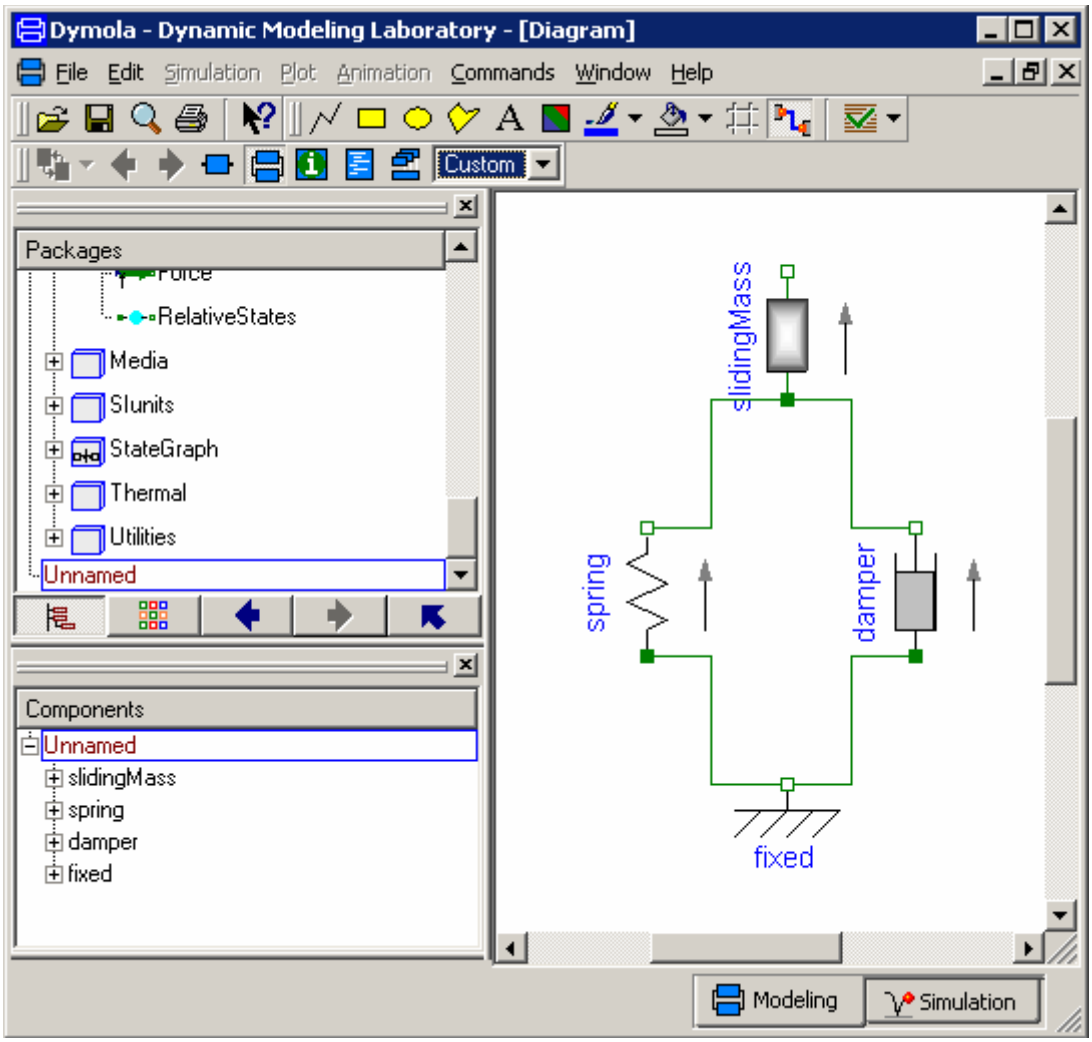

Figure 3.6: A Modelica connection diagram for a MSD CD model.

While Modelica connectors, the flow prefix, and connect clauses are convenient modeling tools, they have no direct equivalents in SysML. This could be resolved through the creation of several SysML extensions via stereotypes, but this greatly restricts the modeling approach outlined in this section (Section 3.3) to the creation of Modelica models in SysML. The approach certainly relies on Modelica as a foundation, but should still be general enough to facilitate the integration of a variety of CD modeling languages. Furthermore, creating SysML extensions for the purpose of hiding the details of a CD model seems to contradict the idea of "white box" modeling.

To describe component interactions in SysML using a “white box” approach, the system nodes must be represented explicitly. System nodes are used to impose common constraints on system parts and don’t necessarily represent system components. To recognize this notion, node definitions should be relegated to constraint blocks. A system model can then own *constraint properties* (usages of constraint blocks) to represent system nodes. Using a SysML *parametric diagram*, the parameters used in the definition of a constraint block can be bound to the properties of another block or constraint block using *binding connectors*. A binding connector implies a *pure equality* constraint between two objects. If the objects are part properties, then all of the sub-properties belonging to each part are equal. Hence, binding the interface of a system component to a parameter of a system node implies that any nested value properties in the component interface are equal to their counterparts in the node parameter. This corresponds to using a Modelica connect clause to connect two interface components that don’t contain flow variables.

Figure 3.7 illustrates the definition of two constraint blocks named *MechNode3* and *MechNode4*.

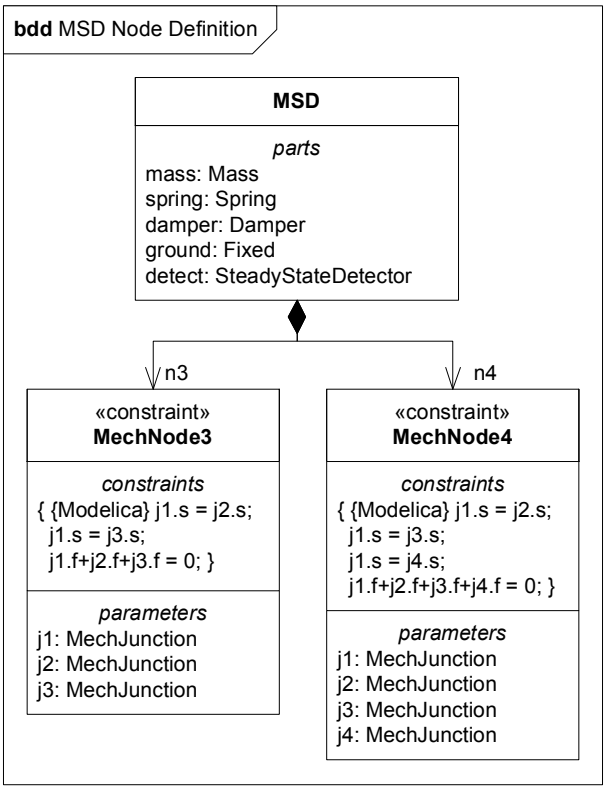

Figure 3.7: Declaration of the mechanical node constraint blocks.

These constraint blocks have several parameters of the type *MechJunction*. The across and through variables of these parameters are subject to the packaged constraints that describe Kirchhoff's circuit laws for a translational mechanical system. MSD owns one usage of each constraint block to enable the interaction of its part properties. Figure 3.8 displays a parametric diagram that depicts the part interactions as a result of binding usages of *MechJunction*. Note the resemblance of Figure 3.8 to Figure 3.1.

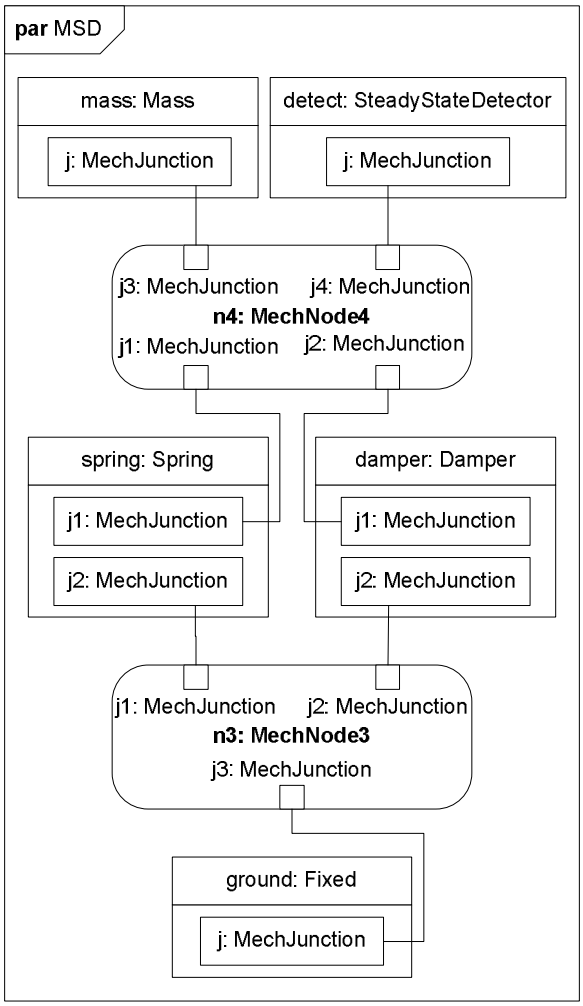

Figure 3.8: The parametric diagram of the *MSD* block.

The Modelica equivalent to Figure 3.7 and Figure 3.8 can be seen in Figure 3.9.

```
//The MSD declaration
class MSD
  //The system components
  Mass mass;
  Spring spring;
  Damper damper;
  Fixed ground;
  SteadyStateDetector detect;
  //The system nodes
  MechNode3 n3;
  MechNode4 n4;
equation
  //The system composition
  connect(mass.j, n4.j3);
  connect(detect.j, n4.j4);
  connect(spring.j1, n4.j1);
  connect(damper.j1, n4.j2);
  connect(spring.j2, n3.j1);
  connect(damper.j2, n3.j2);
  connect(ground.j, n3.j3);
end MSD;

//The MechNode3 declaration
class MechNode3
  MechJunction j1, j2, j3;
equation
  j1.s = j2.s;
  j1.s = j3.s;
  j1.f+j2.f+j3.f = 0;
end MechNode3;

//The MechNode4 declaration
class MechNode4
  MechJunction j1, j2, j3, j4;
equation
  j1.s = j2.s;
  j1.s = j3.s;
  j1.s = j4.s;
  j1.f+j2.f+j3.f+j4.f = 0;
end MechNode4;
```

Figure 3.9: The Modelica representation of a fully composed MSD system model.

3.4 Integrating "Black Box" CD Models into SysML

Oftentimes, engineers reuse existing computational models when solving systems engineering problems. If an engineer wishes to reuse an existing Modelica CD model and integrate it into a larger SysML information model, recreating the model in SysML using the approach outlined in Section 3.3 could prove to be a cumbersome task. In this section, a modeling approach is described for integrating pre-existing, external models into SysML by representing only their most important details and an interface for user

and model interaction. System models can then be composed of these external models using binding connectors and Modelica-specific system nodes.

3.4.1 Model Declaration

When building models using a "white box", high-fidelity modeling approach such as that outlined in Section 3.3, a modeler must declare every detail needed to define completely the model of interest; however, when using a "black box", low-fidelity modeling approach, a modeler only needs to acknowledge sufficiently the referenced model and its most important details.

The first step in referencing an external model is to create a SysML object representing that model. Since the primary SysML modeling unit is the block and the modeling approach outlined in Section 3.3 relies on the use of blocks, the representation of an external model should be relegated to a block; however, using blocks to represent both "white box" and "black box" could be confusing if a modeler can't easily distinguish between both types of blocks.

To identify a "black box" block referencing an external model, the «external» stereotype is introduced to enable SysML modelers to acknowledge dependence upon an external model. This stereotype is an original extension to SysML. When a block is assigned the *«external»* stereotype, the modeler is obliged to include necessary model metadata by adding the value properties *url:String*, *fqn:String*, and *mime:String*. These properties enable the identification and high-level description of the external model. While these properties are sufficient for the work done in this thesis, the *«external»* stereotype could be extended or modified to impose other important metadata. The *url* property takes on the value of the external model's uniform resource locator (URL). This

allows a SysML model transformer to locate the file containing the referenced model. The *fqn* property takes on the value of the referenced model's fully qualified name. This identifies the model location within the file specified by *url*. The *mime* property classifies the referenced model and takes on the value of a descriptive phrase or keyword.

Figure 3.10 demonstrates the declaration of external blocks through the creation of an MSD system model that utilizes "black box" references to four external translational-mechanics models from the Modelica Standard Library (MSL).

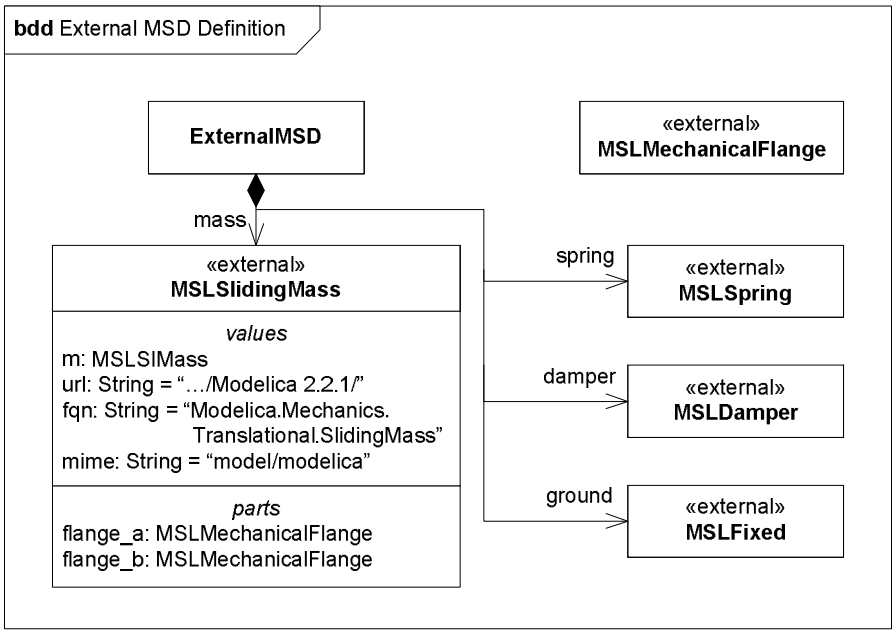

Figure 3.10: The declaration of the *ExternalMSD* SysML CD model.

This BDD is very similar to Figure 3.3 in that a block representing the whole system of interest owns usages of and is decomposed into blocks that describe the subsystems or components. Note that the *ExternalMSD* block is the only block without the «external» stereotype as it does not refer to an existing Modelica model. All of the other blocks do have the stereotype and accordingly own *url*, *fqn*, and *mime* properties with appropriate values. In the case of the *MSLSlidingMass*, its *url* points to the location of the MSL, *fqn*

identifies the actual name of the model in the MSL, and *mime* has the value "model/modelica" to signify that the block is referencing a Modelica model.

A "black box" model is intended to hide details from a model user; however, hiding *all* details is not permissible since a modeler often cares about certain properties in the referenced model. Accordingly, most properties need not be shown in an external block, but those representing model parameters or variables of interest must be exposed to the user. Otherwise, the external block has a limited application base. To recognize and utilize these properties, a user should acknowledge them in an external block by adding value properties that have the same name and type as the actual property in the referenced model. Figure 3.10 demonstrates this modeling approach by acknowledging the parameter *m* owned by the Sliding Mass model in the MSL.

3.4.2 Model Interface

While many unnecessary details are omitted from the declaration of an external block in SysML, the block's interface must be explicitly defined to enable the creation of system models composed of external models. Just as described in Section 3.3.2, the interface for model interaction is declared using part properties typed to blocks containing across and through variables. The major difference however is that when declaring the typed interface blocks, the across and through variables don't need to be shown. Instead, the typed blocks are also assigned the «external» stereotype and given appropriate metadata.

Figure 3.10 demonstrates the declaration of external interface blocks through the depiction of a reference to the MSL Mechanical Flange model commonly used by MSL Mechanics models. The other external blocks in Figure 3.10 contain usages of these

flange blocks using the names of their counterpart Modelica connector components. For example, *MSLSlidingMass* owns usages of *MSLMechanicalFlange* with names *flange_a* and *flange_b* since the MSL Sliding Mass model owns usages of the MSL Mechanical Flange model with the names *flange_a* and *flange_b*.

3.4.3 Composing a System Model

As discussed in Section 3.3.5, CD system models are composed by connecting usages of blocks that represent a system's component or subsystem. In a similar fashion, modelers might need to create a CD system model that relies on connected usages of external blocks. Just as Section 3.3.5 describes the use of system nodes enforcing constraints upon the across and through variables exposed in the interfaces of system parts, the approach to connecting usages of external blocks relies on Modelica-specific system nodes that impose Modelica connect clauses. Connect clauses are used in place of an explicit representation of an equivalent to Kirchhoff's laws because most native Modelica CD models own usages of connectors that employ the flow prefix. Hence, a connect clause that connects two interfaces using the flow prefix implicitly imposes an equivalent of Kirchhoff's laws.

To demonstrate the use of Modelica-specific system nodes, Figure 3.11 displays the declaration of a node constraint block owning a constraint that imposes two Modelica connect clauses on its parameters.

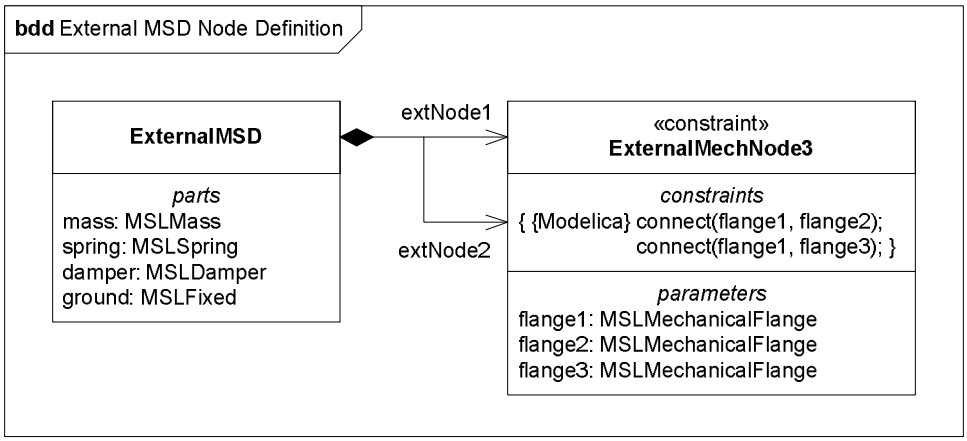

Figure 3.11: Declaration of a constraint block representing a Modelica-specific node.

Each node block has *MSLMechanicalFlange* parameters that are referenced in its constraint(s). Connecting a part's usage of *MSLMechanicalFlange* (e.g. *mass.flange_b: MSLMechanicalFlange*) to a flange belonging to a node in effect substitutes the system component's flange in the connect clause modeled by the node's constraint. To compose a system model, binding connectors are placed between system components and system nodes using the same approach outlined in Section 3.3.5. This is demonstrated in Figure 3.12.

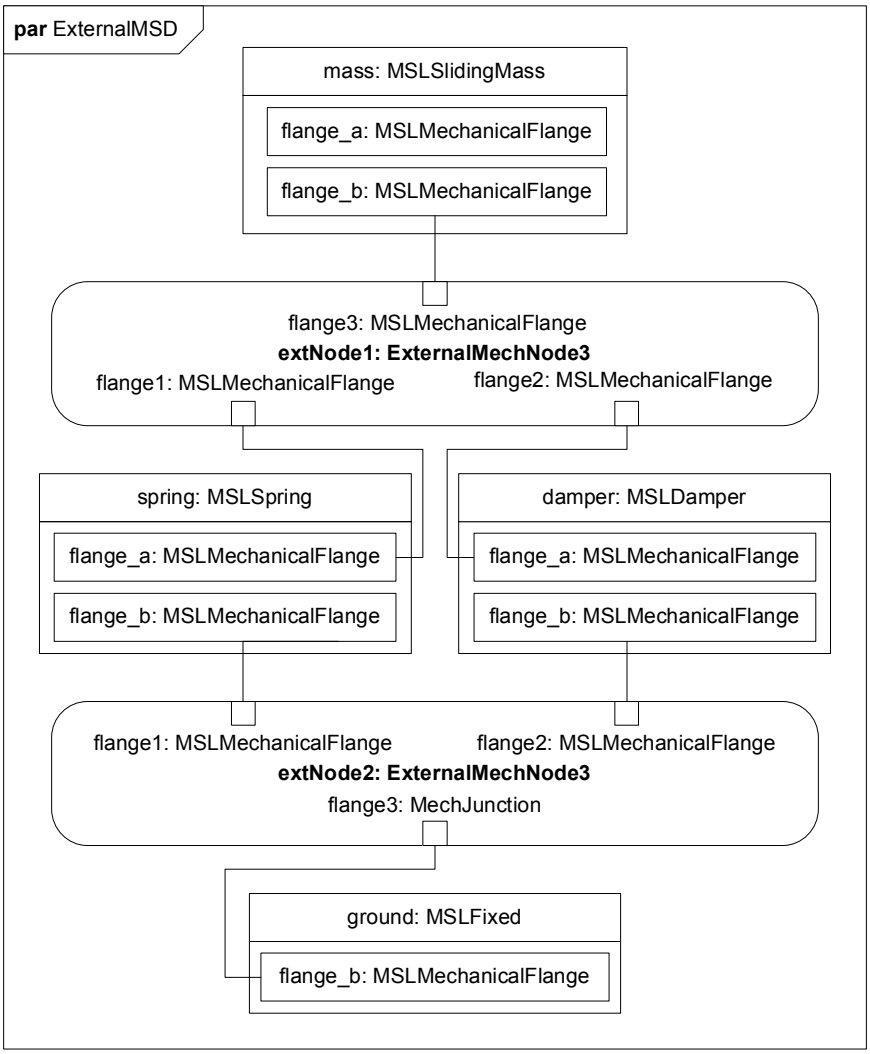

Figure 3.12: The parametric diagram of the *ExternalMSD* block.

While the use of Modelica-specific system nodes explicitly captures Modelica connect clause syntax, it can become cumbersome when composing system models. As discussed in Section 3.3.5, a SysML binding connector maps directly to a Modelica connect clause under the assumption that all variables contained in a SysML model's interface don't rely on an equivalent of the Modelica flow prefix. Hence, any time a SysML-to-Modelica transformer encounters a SysML connector, a Modelica connect clause is created. Consequently, a "hack" of sorts is introduced in which a modeler can substitute simple binding connector(s) in place of a Modelica-specific system node. In

Figure 3.12, *ground.flange_b* is connected to *extNode2.flange3* while *extNode2.flange1* is connected to *spring.flange_b*. The corresponding set of Modelica equations are *connect(ground.flange_b,extNode2.flange3), connect(extNode.flange1,extNode2.flange1)* (this comes from the constraints of the block *ExternalMechanicalNode3*), and *connect(extNode2.flange,spring.flange_b).* This set of Modelica equations can be reduced to *connect(ground.flange_b, spring.flange_b)* which corresponds to a SysML connector placed directly between *ground.flange_b* and *spring.flange_b*. Hence, Modelica-specific system nodes aren't necessary, but their removal from a SysML model portrays incorrect semantics since the binding connector replacements are used to represent the imposition of circuit laws rather than pure equalities.

One option is to leave this modeling practice as a hack that is only effective when dealing with external models that rely on the Modelica flow prefix. Alternatively, the binding connector can be extended using a UML stereotype to ensure that a parametric diagram of a CD model depicts the correct semantics. This original stereotype, named *«connectClause»*, can be applied to a binding connector placed between two part properties typed to external blocks representing Modelica connectors. The semantics of the stereotype state that the binding connector actually represents a Modelica connect clause instead of simply pure equality. Examples of the *«connectClause»* binding connector are displayed in Figure 3.13 and in the excavator model of Chapter 6.

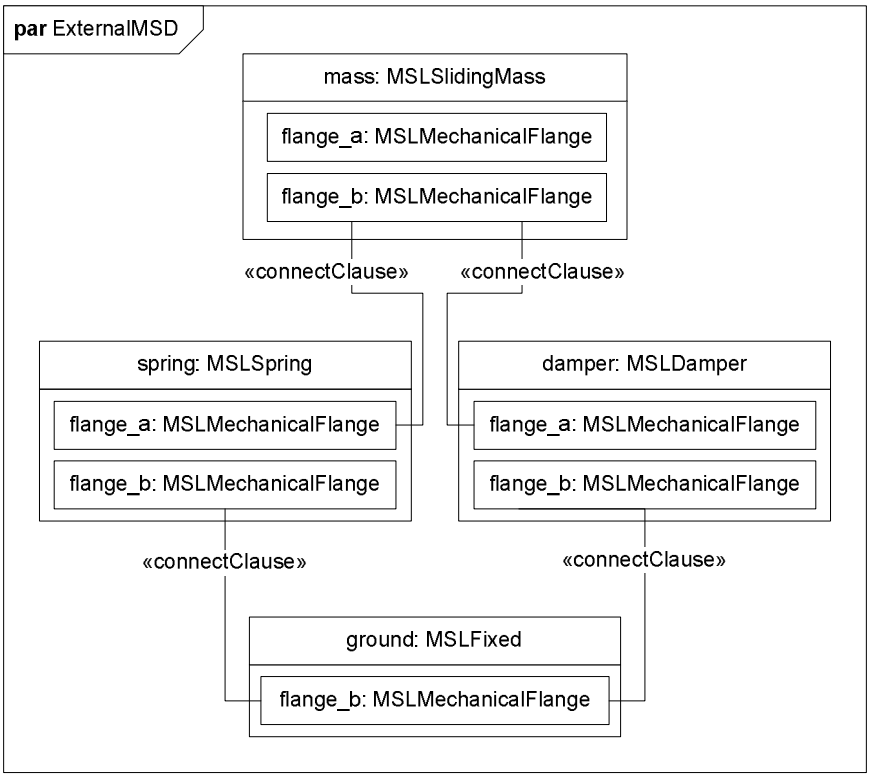

Figure 3.13: Using *«connectClause»* binding connectors in place of system nodes.

3.5 Summary

This chapter outlines in detail the approach to representing CD models using the graphical modeling constructs provided in SysML. Section 3.1 first establishes the objectives of the approach to ensure that its use provides a SysML modeler with a valuable CD modeling ability. Section 3.2 initiates the explanation of the SysML CD modeling approach by providing justification for using Modelica as the foundational CD modeling language. Section 3.3 provides an exhaustive approach to creating fully-detailed, "white box" CD models in SysML. To facilitate the simplification of CD modeling in SysML, Section 3.4 provides a convenient approach to creating "black box" SysML CD models that act as proxies for existing Modelica models.

The SysML CD modeling constructs outlined in this chapter are the foundation for integrating CD models with other SysML models. Using these constructs, modelers can abstract important knowledge from system CD models into SysML such that information can be shared amongst the various other models represented in a SysML information model. Furthermore, the language mapping used to develop the SysML CD modeling approach can be used to develop a graph transformation schema for automating the transformation of information and knowledge between SysML and Modelica models.

CHAPTER 4

TRANSFORMING BETWEEN SYSML AND MODELICA MODELS

In Chapter 3, an approach was described for representing CD models in SysML via a language mapping between SysML and Modelica. One of the objectives of the approach was to enable the transformation of SysML CD models into Modelica models for the purpose of model execution. In this chapter, the language mapping is extended into a graph transformation schema and transformation rules that enable the automation of SysML-to-Modelica model transformations.

4.1 The Need for Graph Transformations

If true model integration is to occur in SysML, engineers must be able to link external models adhering to languages other than SysML to models existing in SysML. Such a linkage permits the creation of dependencies between design and analysis models authored in SysML or in other languages. In the context of CD modeling in SysML, the linkage to Modelica models is partially established by the CD "white box" and "black box" modeling approaches described in Chapter 3; however, the ability to abstract a Modelica CD model into SysML doesn't necessarily provide the ability to affect the Modelica model through the representation of bindings and associations to the SysML model. To provide this ability, a modeler must be able to transform knowledge/information between SysML and Modelica models. Preferably, these transformations are automated to ensure fast and error-free transformations.

One option for automating the transformation process is by using a typical computer programming language (e.g. Java, C/C++) to create software that is able to query and transform SysML and Modelica models through the use of large, complex sets of logical constructs (e.g. switch statements, if statements). While this is a feasible approach to implementing model transformations, it might not be the most user-friendly and adaptable approach.

Alternatively, another option for automating the transformation process is through the use of a higher-level approach that is better suited for implementing model transformations. One such high-level approach is the use of graph transformations. Instead of using complex sets of low-level logic, graph transformations rely on pattern matching abilities built into graph transformation tools (e.g. VIATRA) to identify precondition patterns in a source model and to prescribe postcondition patterns in a target model. In the context of SysML-Modelica transformations, graph transformations can be used to locate and specify patterns in a graph of a SysML or Modelica model.

Outside of the relative ease of incorporating graph transformations, another important benefit is the preservation of graph patterns between source and target models. When performing graph transformations, the resultant graph can be preserved and reused for future propagations of changes in a source or target model. This is not easily accomplished using low-level logical constructs.

Overall, graph transformations provide a convenient mechanism for completing model transformations. The implementation of graph transformations for the purpose of transforming SysML and Modelica models provides the following potential functionality: the generation of Modelica models from SysML models and vice-versa; and the

propagation of changes in Modelica models to SysML models and vice-versa. When these abilities are obtained, true "execution" links can be established between SysML and Modelica models.

4.2 The Transformation Approach

Due to the benefits of performing model transformations with TGGs, the transformation approach outlined in this chapter revolves around the creation of a TGG and corresponding operational graph transformation rules. Operational graph transformation rules are scenario-specific rules for transforming source modeling elements into corresponding target modeling elements. In contrast, actual TGG graph transformation rules are declarative by nature and more powerful since they enable bidirectional model transformation and model synchronization; however, these rules are difficult to implement because not all transformations are bidirectional and many model transformation tools are not capable of executing bidirectional transformation rules. In this chapter, operational rules are developed for performing SysML-to-Modelica transformations because they sufficiently demonstrate the power of graph transformations and their potential for improving MBSE. Moreover, the TGG described in this chapter can still facilitate the development of actual TGG rules.

4.2.1 The SysML and Modelica Metamodel Subgraphs

The key to developing a TGG is the language mapping. By examining the mapping in detail, the essential modeling elements from each language can be identified and separated from the non-essential elements. For example, a clear mapping exists between SysML blocks and Modelica classes, so both elements must be acknowledged in

the SysML-to-Modelica transformation schema. In contrast, a clear mapping does not exist between the Modelica flow prefix and a SysML modeling construct, so the flow prefix is not included in the transformation schema. Once the necessary modeling elements are identified, graph-based representations of each language are developed as subgraphs of the TGG. These subgraph metamodels are not intended to represent a modeling language in its entirety; instead, they are incomplete representations enabling model transformations that adhere to the language mapping of interest.

Figure 4.1 displays the subgraph of the SysML metamodel used in the SysML-to-Modelica TGG.

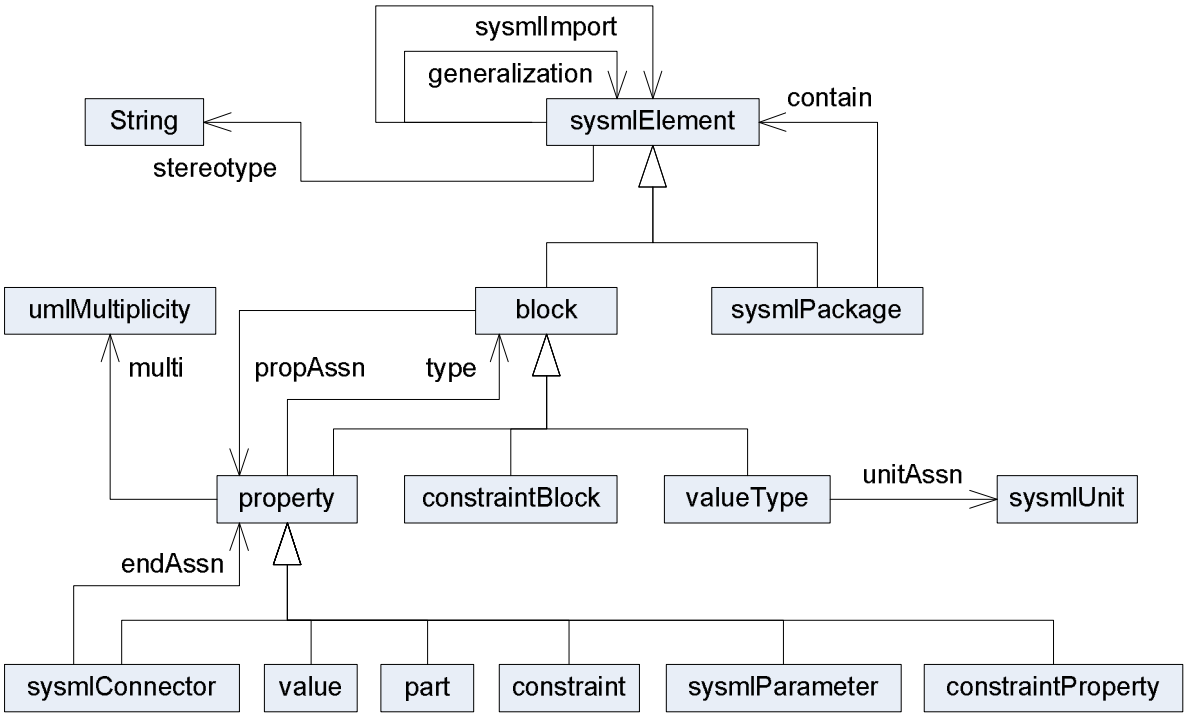

Figure 4.1: The SysML metamodel subgraph of the SysML-to-Modelica TGG.

This representation of the SysML metamodel strikes a compromise between maintaining accuracy and fostering ease of use. Additionally, modeling elements that are not required in the SysML-to-Modelica transformation are excluded (e.g. requirements). Important modeling elements such as blocks, packages, properties, and connectors are included

while unnecessary elements like connector ends and roles are replaced with the simple relation *endAssn* pointing from a connector to a property.

Figure 4.2 displays the graph of the Modelica metamodel used in the development of the SysML-to-Modelica TGG.

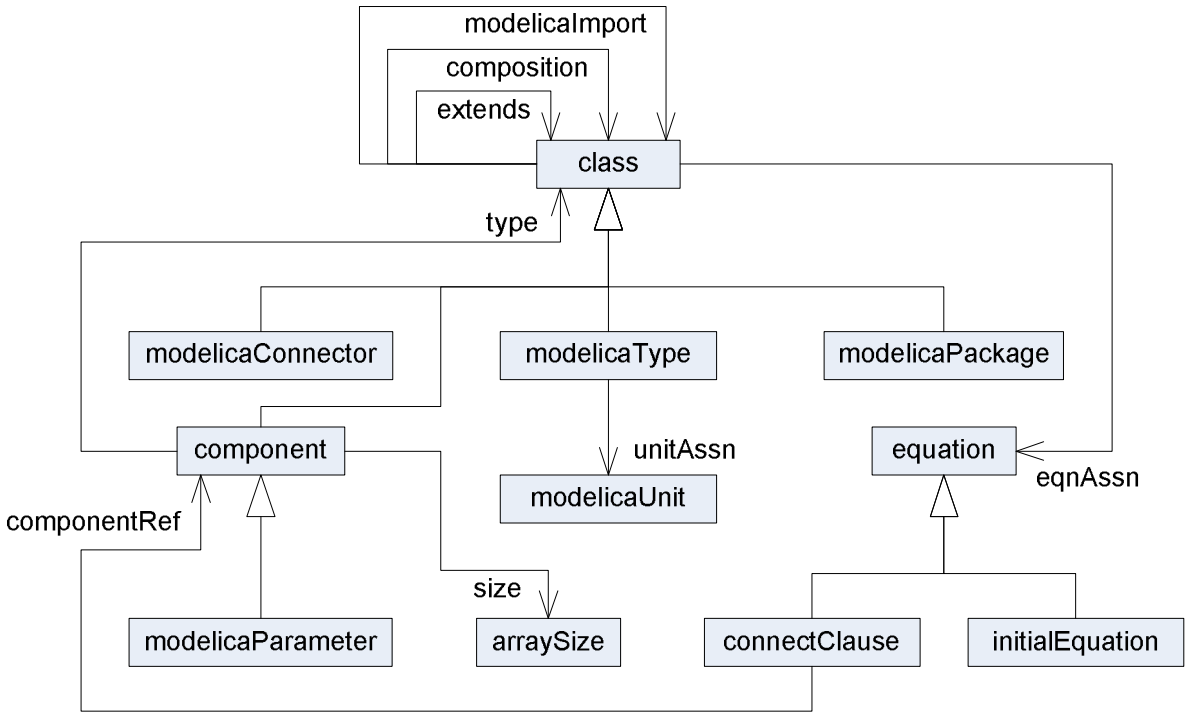

Figure 4.2: The Modelica metamodel subgraph of the SysML-to-Modelica TGG.

Again, the intent of this graph is not to reflect directly the Modelica language specification [11], but to strike a balance between accuracy and ease of use.

4.2.2 The Correspondence Metamodel Subgraph

To develop the correspondence graph for the TGG, each mapping described in Chapter 3 is translated into a correspondence element that points to the mapped elements. This results in the specification of the SysML-to-Modelica TGG as depicted in Figure 4.3.

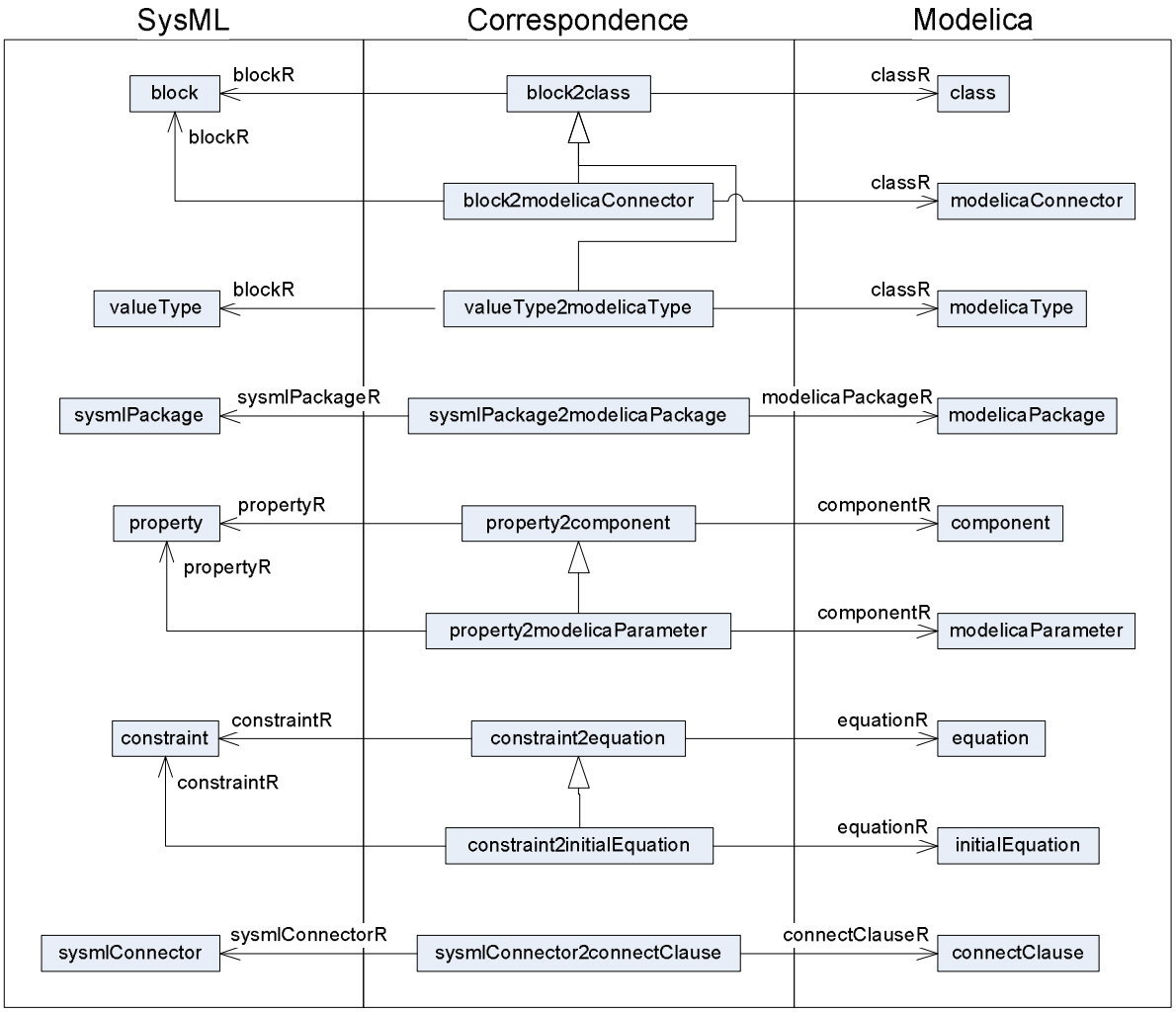

Figure 4.3: The Correspondence metamodel subgraph of the SysML-to-Modelica TGG.

Most correspondence modeling elements have been defined only if they were necessary for ensuring model traceability. A complete correspondence metamodel would include correspondence elements for every mapping between SysML and Modelica. For example, the correspondence between a block and a class was deemed necessary while a correspondence between UML multiplicities and Modelica array sizes was deemed unnecessary.

4.2.3 The Graph Transformation Rules

When the TGG is complete, operational graph transformation rules can be developed that force a source and target model to satisfy the TGG. As depicted in Figure 2.2, graph transformations are used to read a source model adhering to a source metamodel and write a corresponding target model adhering to a target metamodel. In the context of TGGs, a specific sequence of operational graph transformation rules is used to search through source, target, and correspondence graphs to match a given precondition pattern. When the precondition pattern is satisfied, a postcondition pattern that satisfies the TGG is prescribed resulting in the creation of new correspondence and target modeling elements.

In the SysML-to-Modelica graph transformation approach, a graph containing instances of SysML metamodel elements is first parsed to identify all top-level (i.e. non-contained) definition modeling elements (blocks, packages, value types, and units). When a top-level definition element is found, instances of the appropriate correspondence element and Modelica metamodel element are created and correspondence relationships are defined. This is depicted in Figure 4.4 through Figure 4.6 (minus some details).

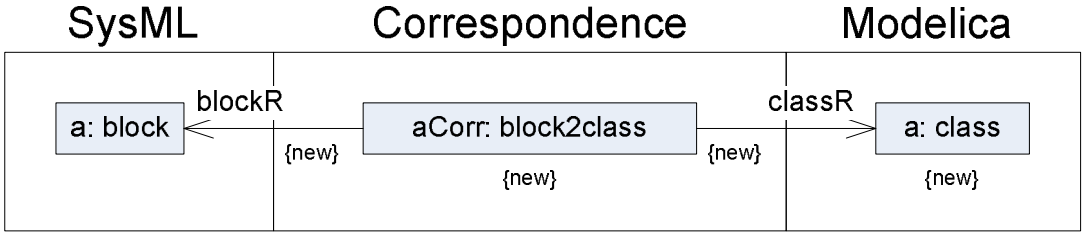

Figure 4.4: The TopBlock-to-Class transformation rule.

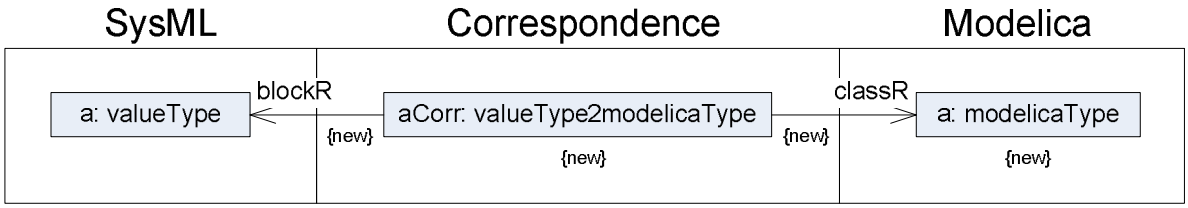

Figure 4.5: The TopValueType-to-ModelicaType transformation rule.

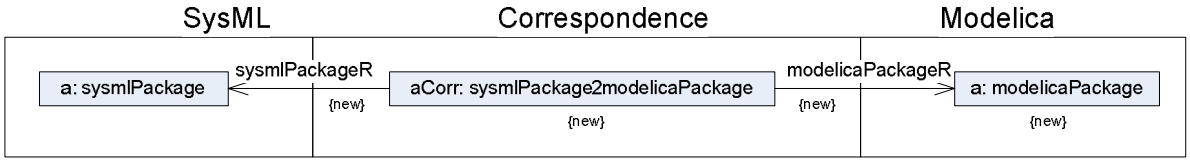

Figure 4.6: The TopSysMLPackage-to-ModelicaPackage transformation rule.

While most top-level definition element transformation rules are simple, a special rule is used to transform SysML blocks into Modelica connectors. As depicted in Figure 4.7, this rule states that instances of *modelicaConnector* and *block2modelicaConnector* correspondence elements should exist if a *block* is used by a *part* that is the target of a *sysmlConnector*'s *endAssn* relationship.

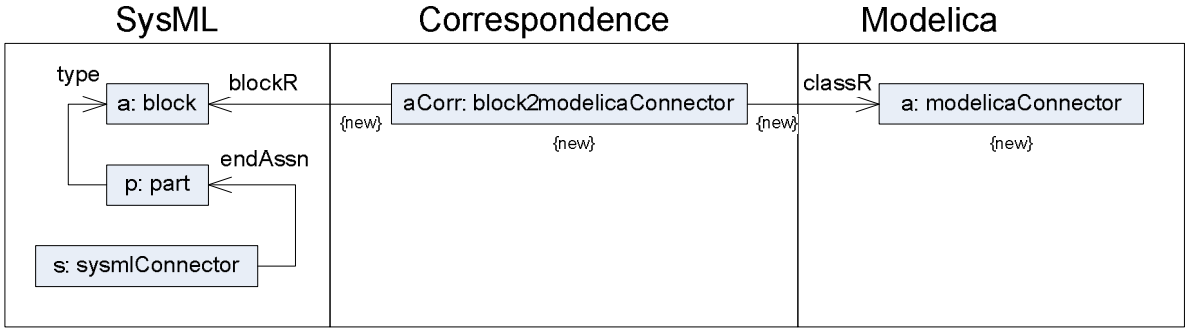

Figure 4.7: The TopBlock-to-ModelicaConnector transformation rule.

Once all instances of top-level SysML definition elements are transformed into their Modelica counterparts, the transformation rule depicted in Figure 4.8 is applied to the SysML model to transform contained *blocks* into contained *classes*.

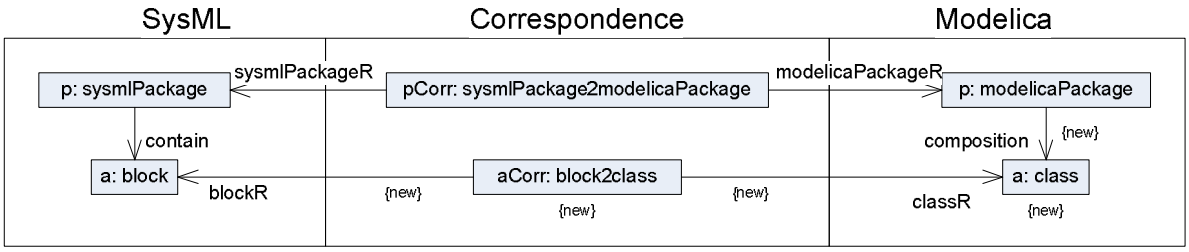

Figure 4.8: The ContainedBlock-to-Class transformation rule.

Similar rules exist for contained *valueTypes*, *sysmlPackages*, and *units*.

The last definition elements that are subject to transformation are blocks stereotyped by the "external" keyword. As depicted in Figure 4.9, the transformation is the nearly identical to that depicted in Figure 4.4, but the resulting class is flagged such that a Modelica code exporter (see Section 4.4) doesn't try to create new Modelica classes that represent existing Modelica classes.

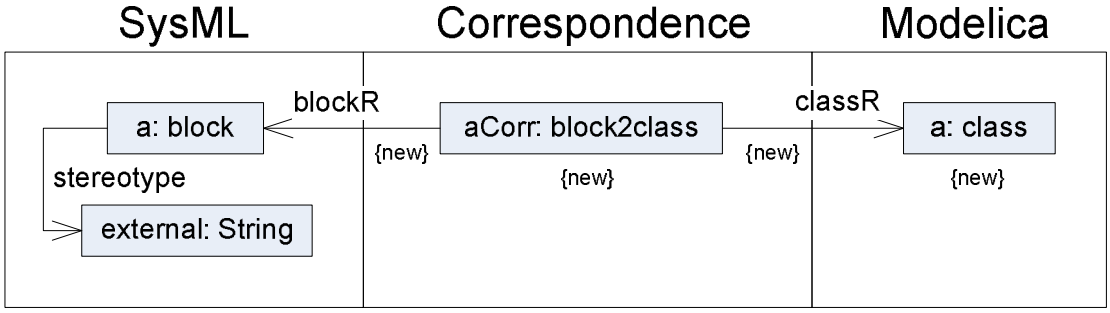

Figure 4.9: The ExternalBlock-to-Class transformation rule.

Once all of the instances of SysML definition elements are transformed, rules are applied to transform SysML *properties*. The general Property-to-Component rule is depicted in Figure 4.10, but specialized rules also exist for transforming specific subtypes of SysML properties.

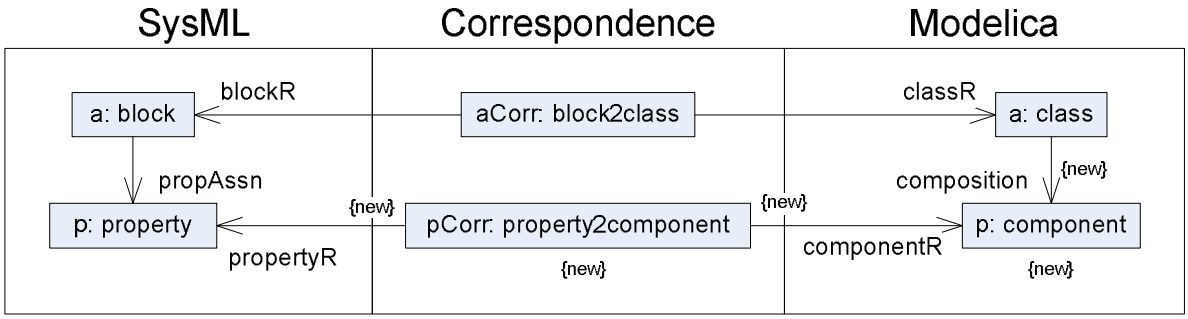

Figure 4.10: The Property-to-Component transformation rule.

The entity structure of the Property-to-Component transformation rule is very similar to that of the ContainedBlock-to-Class transformation rule, but the relationships and element instances have changed. Instead of searching for an instance of a *sysmlPackage* containing an instance of a *block*, the rule searches for an instance of a *block* associated with an instance of a *property*. This structure is also present in the Constraint-to-Equation rule displayed in Figure 4.11 and specialized by the Constraint-to-InitialEquation rule displayed in Figure 4.12.

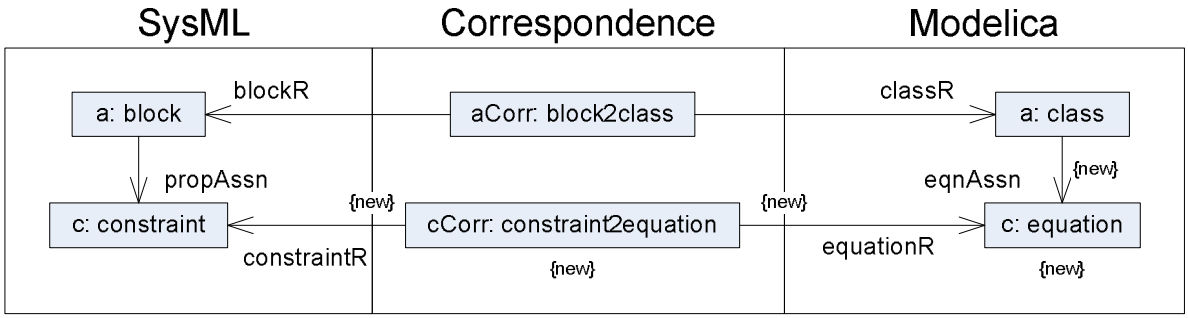

Figure 4.11: The Constraint-to-Equation transformation rule.

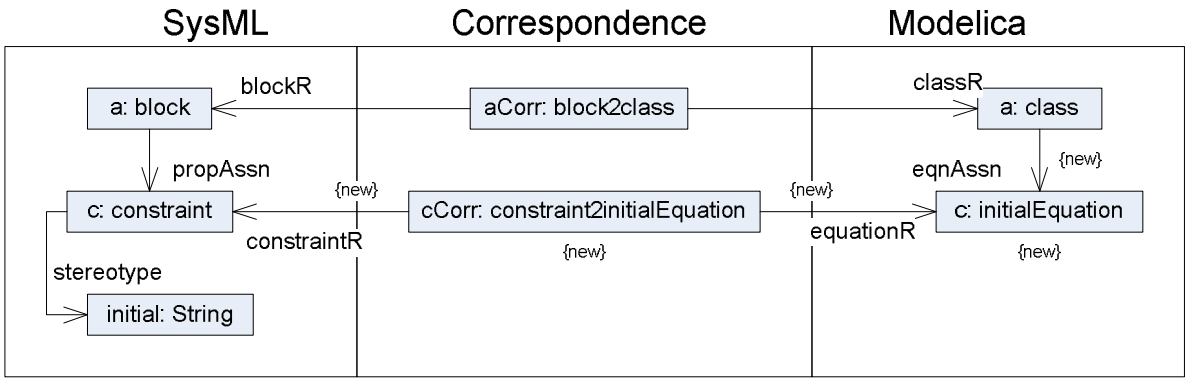

Figure 4.12: The Constraint-to-InitialEquation transformation rule.

Upon finishing the transformation of properties and constraints, SysML connectors that connect two block properties are transformed into corresponding Modelica connect clauses. By searching for *sysmlConnector*s that are the source of two *endAssn*s targeted at two different properties, the transformation rule can create a *connectClause* that directs a *componentRef* relation to the two appropriate *components*. This transformation rule is depicted in Figure 4.13.

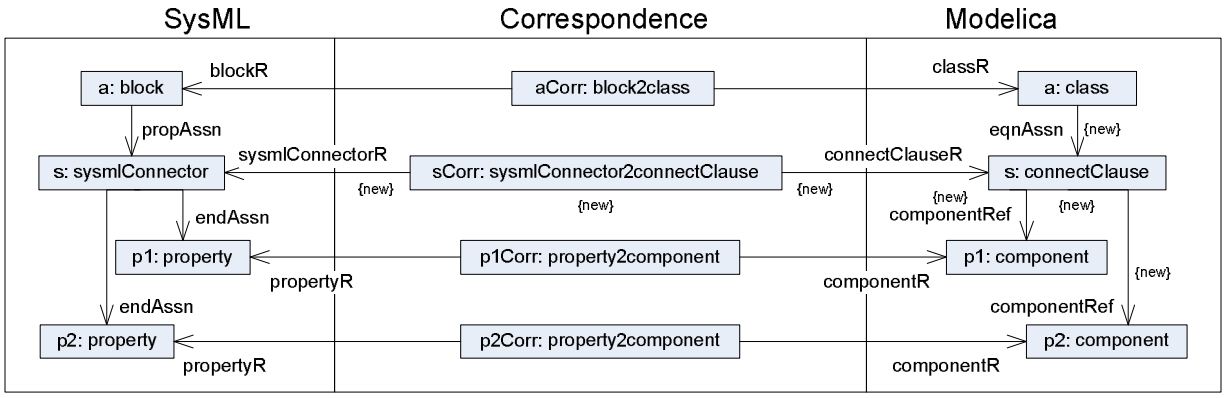

Figure 4.13: The SysMLConnector-to-ConnectClause transformation rule.

After all SysML connectors have been transformed into Modelica connect clauses, the *sysml2modelica* machine finishes the model transformation by transforming simple SysML constructs like the abstract construct, import association, generalization association, and UML multiplicities.

4.3 SysML-to-Modelica Transformations with VIATRA

To implement the SysML-to-Modelica model transformation approach, the TGG and operational graph transformation rules were encoded using the VIATRA [16, 17] plugin for Eclipse. The VIATRA framework was developed to provide general-purpose support for completing model transformations between various engineering domains and modeling languages. Additionally, it was designed to support many transformation standards including OMG's Query/View/Transformation (QVT) [37]. VIATRA is comparable to other model transformation tools such as Fujaba [42] or MOFLON [43], but offers unique features like recursive patterns and negative patterns with arbitrary negation depths.

To capture patterns, models, and metamodels, VIATRA relies on its own declarative modeling language called the VIATRA Textual Metamodeling Language (VTML). VTML provides two main constructs for representing models or metamodels: *entities* and *relations*. An entity represents a modeling concept (e.g. block, property) while a relation represents a relationship between entities (e.g. property association between a block and a property).

Using this entity-relation concept, the metamodels depicted in Figure 4.1 through Figure 4.3 were described in VTML to create the SysML-to-Modelica TGG. Excerpts of the VTML metamodels can be seen in Figure 4.14.

```
//The SysML Metamodel
entity(SysML) {
  //The SysML block
  entity(block) {
    relation(propAssn, block, property);
    multiplicity(propAssn, one_to_one);
    isAggregation(propAssn, true);
    …
  }
  //The SysML Property
  entity(property) {
    relation(type, property, block);
    …
  }
  …
}
//The Modelica Metamodel
entity(Modelica) {
  //The Modelica class
  entity(class) {
    relation(composition, class, component);
    …
  }
  //The Modelica component
  entity(component) {
    relation(type, component, class);
    …
  }
}
//The Correspondence Metamodel
entity(Correspondence) {
  //The block2class correspondence
  entity(block2class) {
    relation(blockR, block2class, SysML.block);
    relation(classR, block2class, Modelica.class);
  }
  //The property2component correspondence
  entity(property2component) {
    relation(propertyR, property2component, SysML.property);
    relation(componentR, property2component, Modelica.component);
  }
  …
}
```

Figure 4.14: An excerpt of the SysML-to-Modelica TGG as represented in VTML.

As seen in Figure 4.14, the primary modeling elements in VTML are the entity and the relation. For clarification, when specifying a relation the first argument is the relation name, the second argument is the source entity type, and the third argument is the target entity type. For example, a *block* can have a relation *propAssn* pointing from a *block* (preferably itself) to a *property*.

To specify model transformations performed using abstract state machines and graph transformation rules, VIATRA relies on its own imperative command language called the VIATRA Textual Command Language (VTCL). The VTCL language provides a user with several general-purpose constructs used to compute graph transformations. The first construct is the *machine*. A machine can contain a main rule and various other rules (i.e. functions) that perform actions on the elements existing in a VIATRA modelspace. A machine can also contain graph patterns written in VTML syntax that are used to perform pattern matching in a VIATRA modelspace. For a machine to perform graph transformations, VTCL employs a special rule appropriately named the *graph transformation rule* (GTR) that can contain precondition, postcondition, and action sections. The precondition section is written in VTML syntax and used to specify a pattern that must be matched somewhere in the modelspace. The postcondition pattern is also written in VTML syntax used to prescribe how the modelspace should be changed once the precondition is satisfied. After a precondition and postcondition are satisfied, a GTR can use the auxiliary action section to perform a set of imperative actions on the modelspace (e.g. renaming entities and resetting entity values).

Using VTCL, a machine named *sysml2modelica* was developed for performing SysML-to-Modelica model transformations. Excerpts of this machine can be seen in Figure 4.15.

```
//Importing the TGG metamodel
import SysML;
import Modelica;
import Correspondence;
//The sysml2modelica VTCL machine
machine(sysml2modelica) {
  …
  //The property2component graph transformation rule
  gtrule property2componentRule(inout P) = {
    //The precondition pattern required to do transformation
    precondition pattern lhs(B, P, PAssn, C, BCCorr, BR, CR) = {
      block(B) {
        property(P);
        block.propAssn(PAssn, B, P);
      }
      block2class(Corr);
      block2class.blockR(BR, Corr, B);
      block2class.classR(CR, Corr, C);
      class(C);
    }
    //The resulting postcondition pattern
    postcondition pattern rhs(P, PAssn, A, Comp) = {
      block(B) {
        property(P);
        block.propAssn(PAssn, B, P);
      }
      block2class(BCCorr);
      block2class.blockR(BR, BCCorr, B);
      block2class.classR(CR, BCCorr, C);
      property2component(PACorr);
      property2component.propertyR(PR, PACorr, P);
      property2component.componentR(AR, PACorr, A);
      class(C) {
        component(A);
        class.composition(Comp, C, A);
      }
    }
    //Renaming A and Comp and resetting the value of A
    action {
      rename(A, name(P));
      rename(Comp, name(PAssn));
      setValue(A, value(P));
  }
  //The gtrule execution sequence
  rule main() = seq {
    …
    forall P apply property2componentRule(P);
    …
   }
}
```

Figure 4.15: An excerpt of the *sysml2modelica* machine as represented in VTCL.

The machine is divided into two important sections: a set of GTRs that reflect the graph transformation rules described in Section 4.2.3 and a main rule that prescribes the sequence in which the GTRs should be performed. When a user runs the *sysml2modelica*

machine, the GTRs are applied to all SysML elements existing in a specific transformation workspace belonging to a VIATRA modelspace.

Figure 4.16 through Figure 4.18 demonstrate the results of running the *sysml2modelica* machine on an example VIATRA representation of a SysML model.

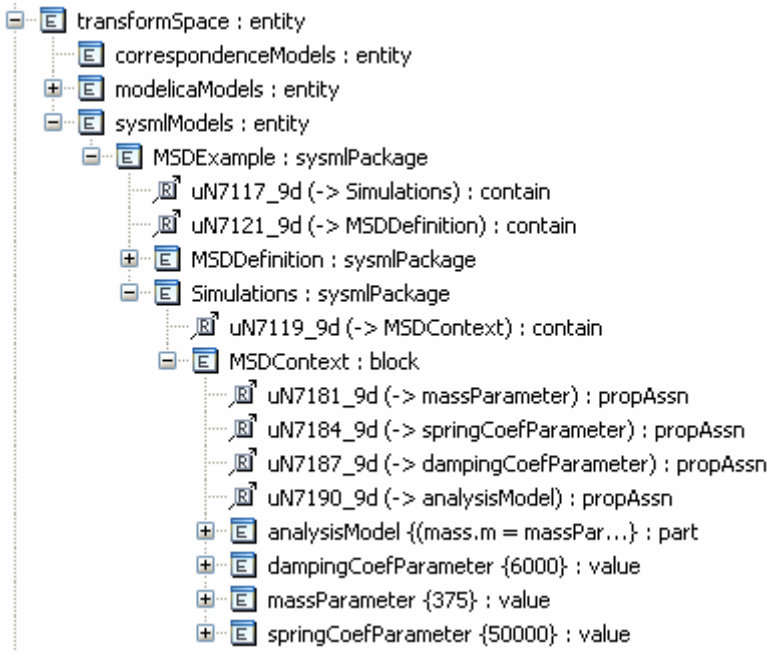

Figure 4.16: A VIATRA representation of a SysML model.

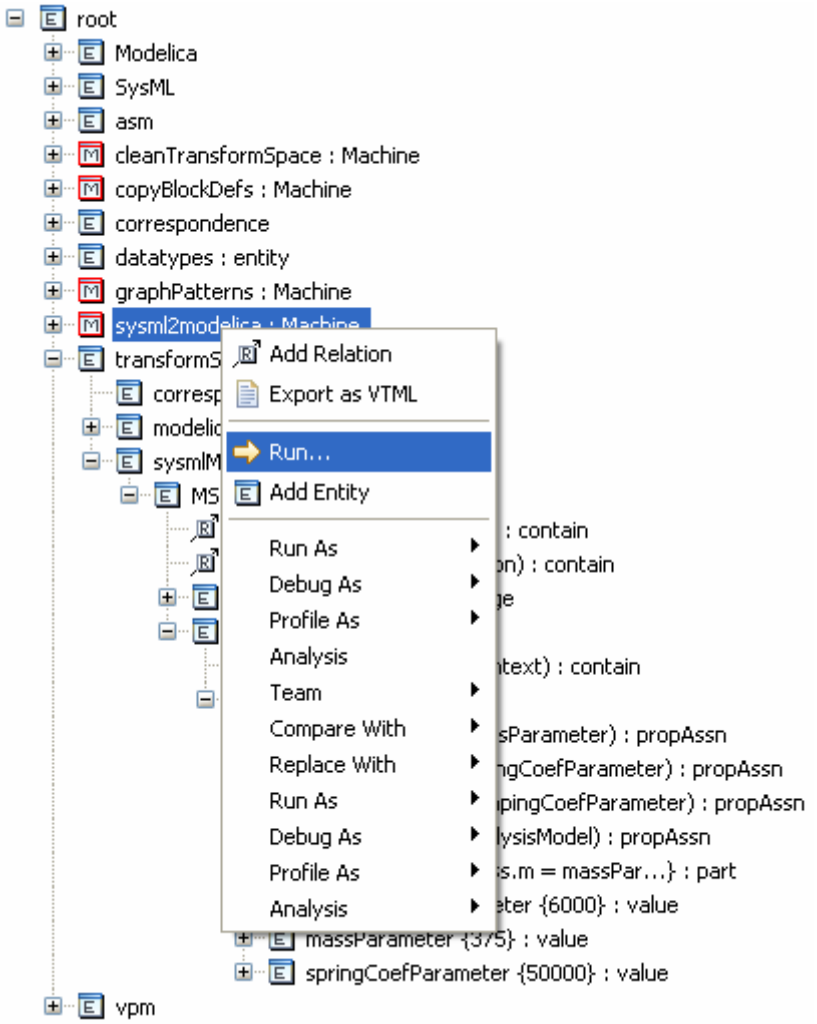

Figure 4.17: Running the *sysml2modelica* machine.

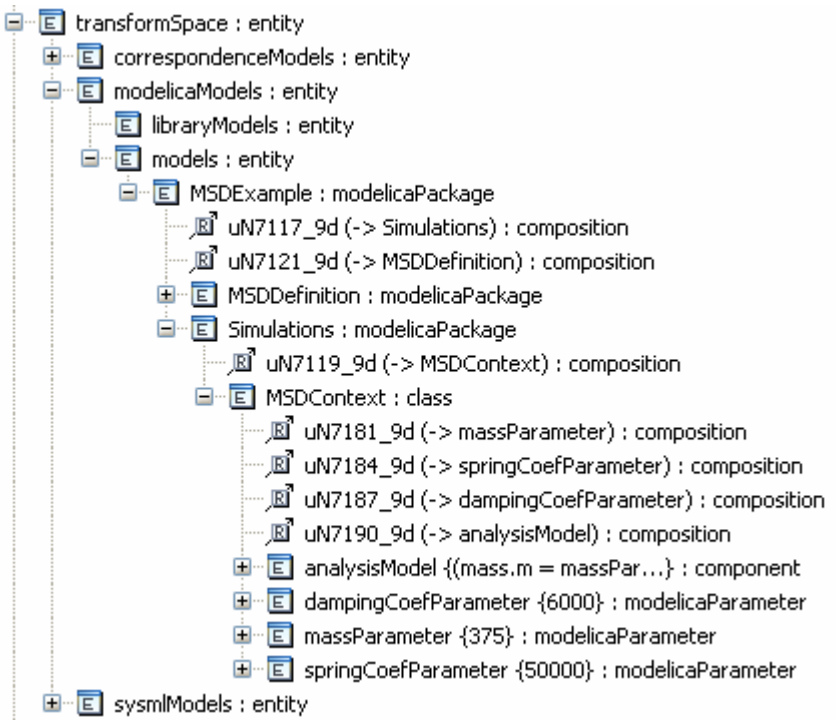

Figure 4.18: VIATRA modelspace resulting from running the *sysml2modelica* machine.

For more information about the VIATRA source code, the best resource is the documentation in the code itself. Most every aspect of the code is well documented using an easy-to-read commenting scheme. The code can be obtained by contacting the author and obtaining the SysMLTransformers plugin [44] and source files.

4.4 Implementation in RSD

In this section, an overview is provided for the SysMLTransformers plugin for the EmbeddedPlus (E+) SysML Toolkit [45] and IBM's extended version of Eclipse called Rational Systems Developer (RSD) [19]. This plugin is used to transform a visual E+ SysML CD model into a lexical Modelica model using VIATRA and the *sysml2modelica* machine. Only the most important classes and details are discussed in the following sections. For more information about the Java source code, the best resource is the

documentation in the code itself. Again, this code can be found in SysMLTransformers plugin and source files.

The plugin source code is divided amongst the nine classes seen in Figure 4.19.

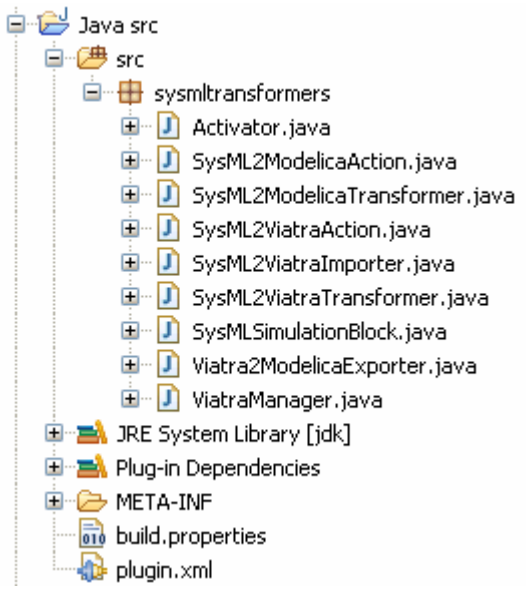

Figure 4.19: The project explorer view of the SysMLTransformers Java source code.

The classes *Activator*, *SysML2ModelicaAction*, and *SysML2ViatraAction* all deal with activating the plugin classes via the RSD project explorer's pop-up menu. The class *SysMLSimulationBlock* is used to store and pass along the properties of a SysML simulation model (Chapter 5). To ease interaction with the VIATRA Application Programming Interface (API), the *ViatraManager* class is used to provide original utility methods and to access commonly used VIATRA API methods for manipulating a VIATRA modelspace. The importer class *SysML2ViatraImporter* is used to access the E+ API and translate a selected E+ SysML CD model into VIATRA syntax. The exporter class *Viatra2ModelicaExporter* to access a VIATRA modelspace and generate Modelica code from a VIATRA representation of a Modelica model. The *SysML2ViatraTransformer* class packages an instance of the *SysML2ViatraImporter* in a

fashion that enables easy execution from the project explorer pop-up menu. The *SysML2ModelicaTransformer* class is very similar to the *SysML2ViatraTransformer* class but is used to do a complete transformation of an E+ SysML CD model using instances of both the *SysML2ViatraImporter* and *Viatra2ModelicaExporter* classes. The functionality of the *SysML2ModelicaTransformer* class is illustrated in Figure 4.20.

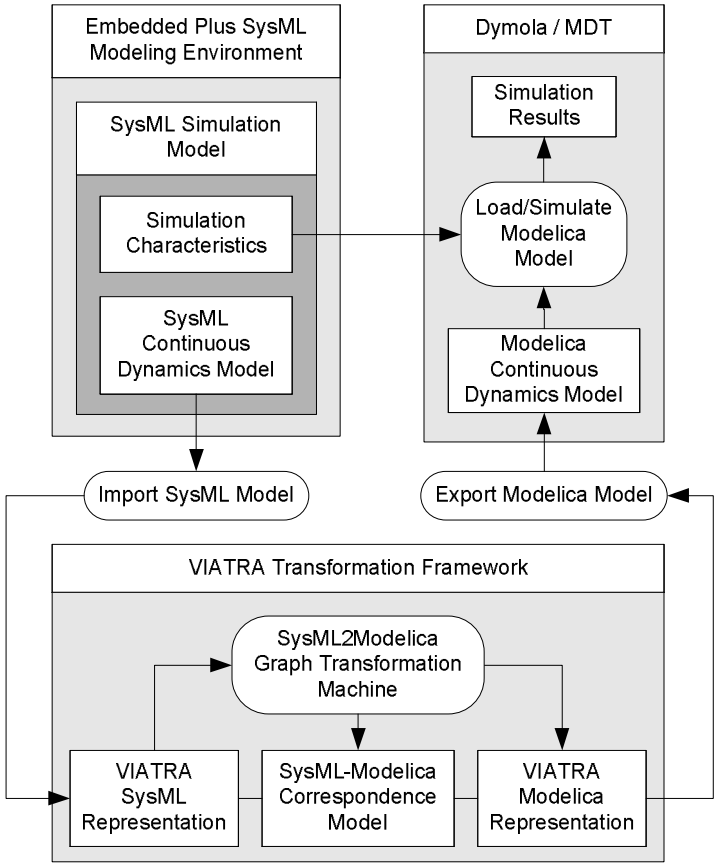

Figure 4.20: The functionality of *SysML2ModelicaTransformer*.

Figure 4.21 through Figure 4.25 illustrate the results of transforming an E+ MSD model by running the *SysML2ModelicaTransformer* through RSD's project explorer pop-up menu. Figure 4.21 shows a BDD of an E+ version of the MSD model that is embedded inside of a SysML simulation model via a model context (Section 5.1 and Section 5.2).

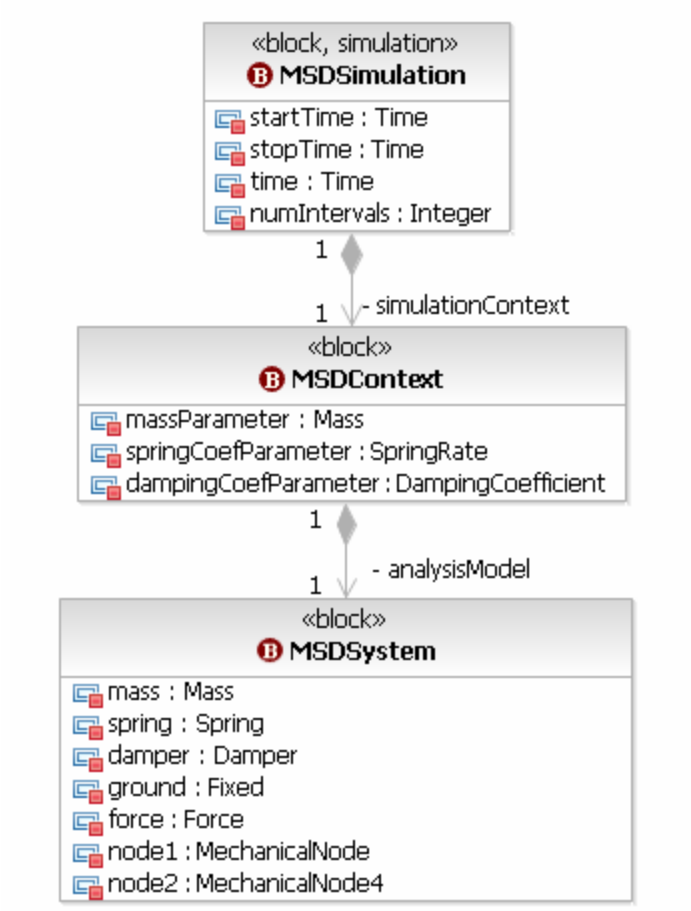

Figure 4.21: A BDD of the E+ *MSDSystem*.

MSDSystem is owned by *MSDContext* which has three value properties characterizing the *mass*, *spring*, and *damper* part properties of *analysisModel*. These properties, *massParameter* (set to 375 kilograms), *springCoefParameter* (set to 50,000 Newtons per meter), and *dampingCoefParameter* (set to 6,000 Newton-seconds per meter), are intended to represent realistic characteristics of a car suspension. Figure 4.22 displays a parametric diagram of *MSDSystem* that is similar to the diagram shown in Figure 3.8.

Figure 4.22: An E+ SysML CD model of a MSD system.

As depicted in Figure 4.23, this model can be transformed into a corresponding Modelica model by right clicking it in the RSD project explorer and selecting “Generate Modelica Model…”.

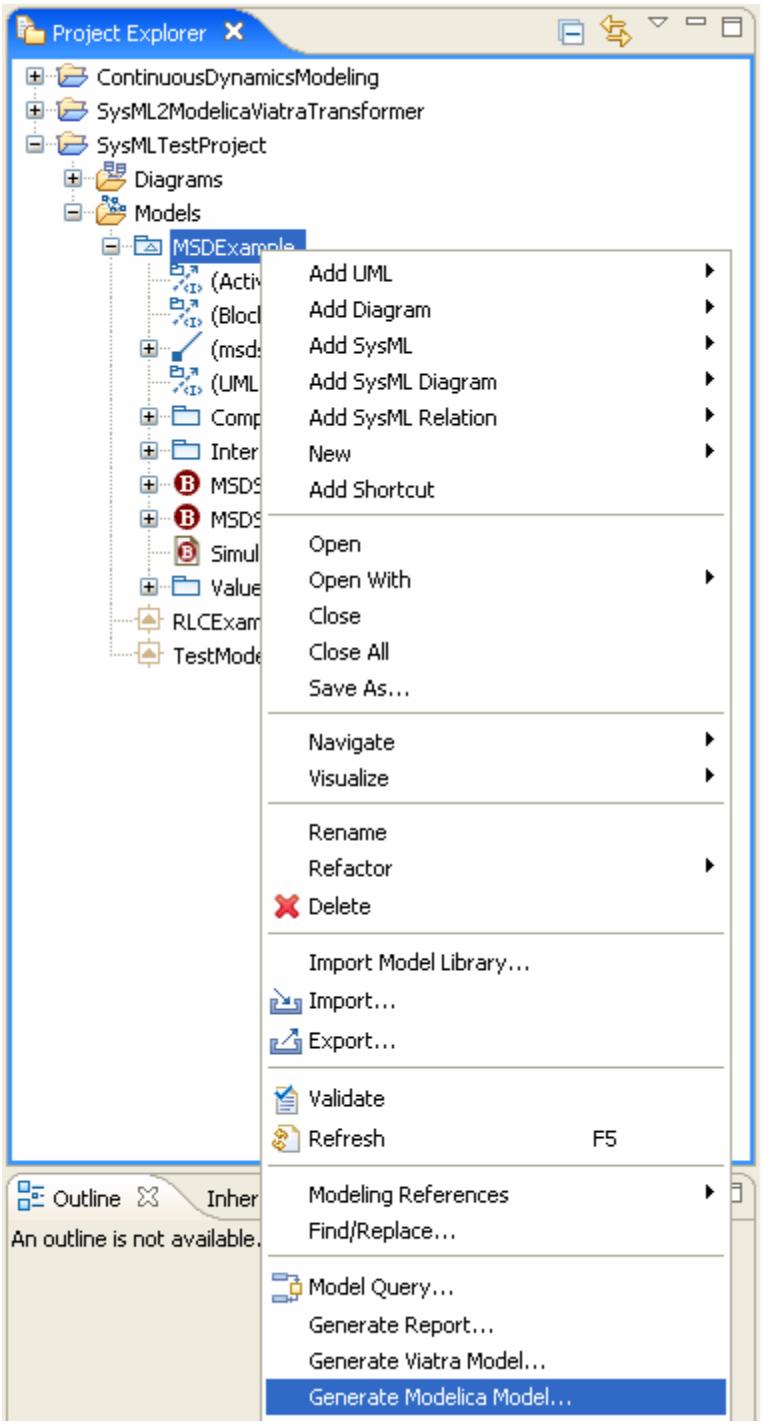

Figure 4.23: Generating a Modelica model from the E+ SysML CD model of a MSD system.

When the *SysML2ModelicaTransformer* completes the transformation process, the resulting Modelica model is placed in a Modelica Development Tooling (MDT) [46] project and imported into Dymola [47] for simulation. This is shown in Figure 4.24 and Figure 4.25.

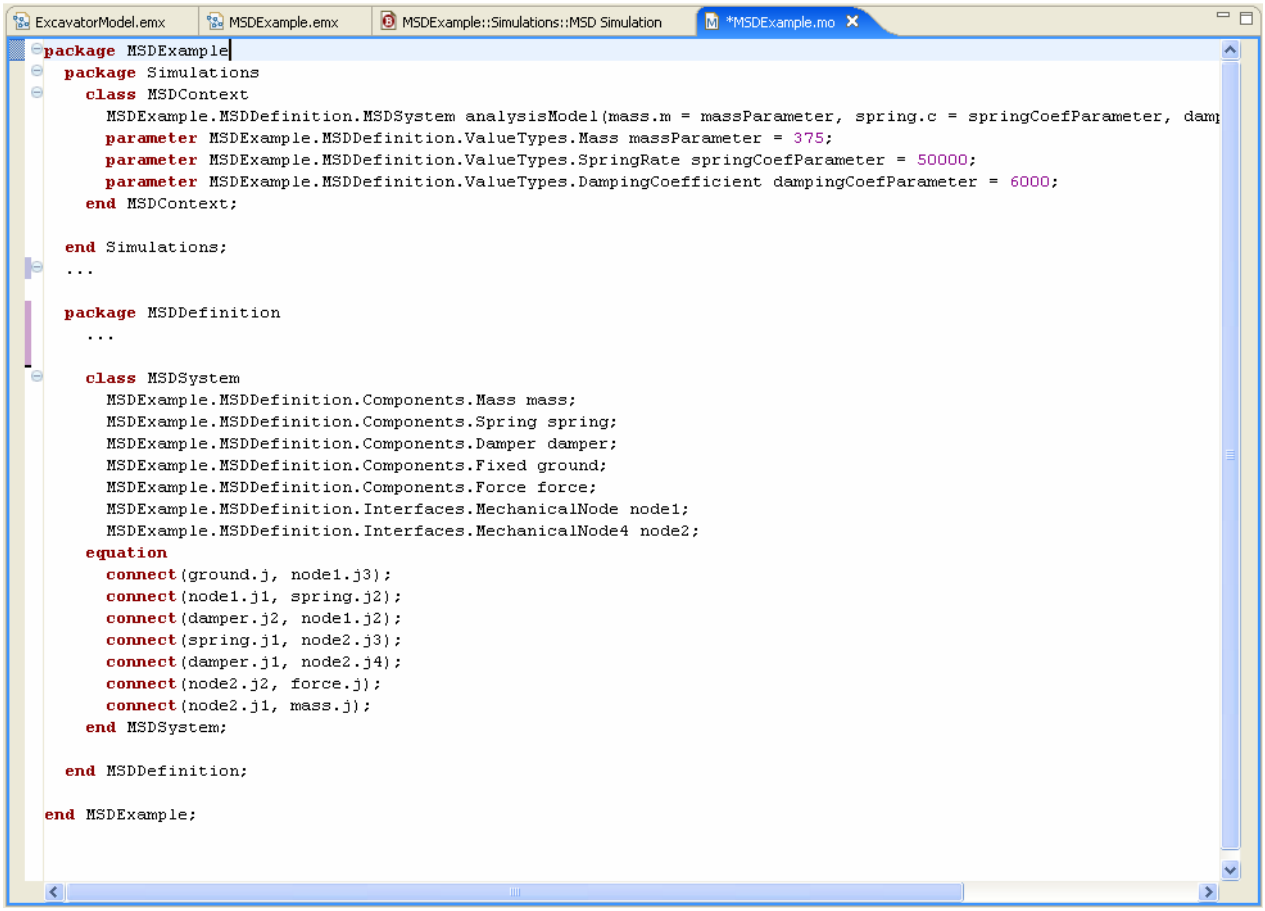

Figure 4.24: An MDT view of the resultant Modelica MSD model.

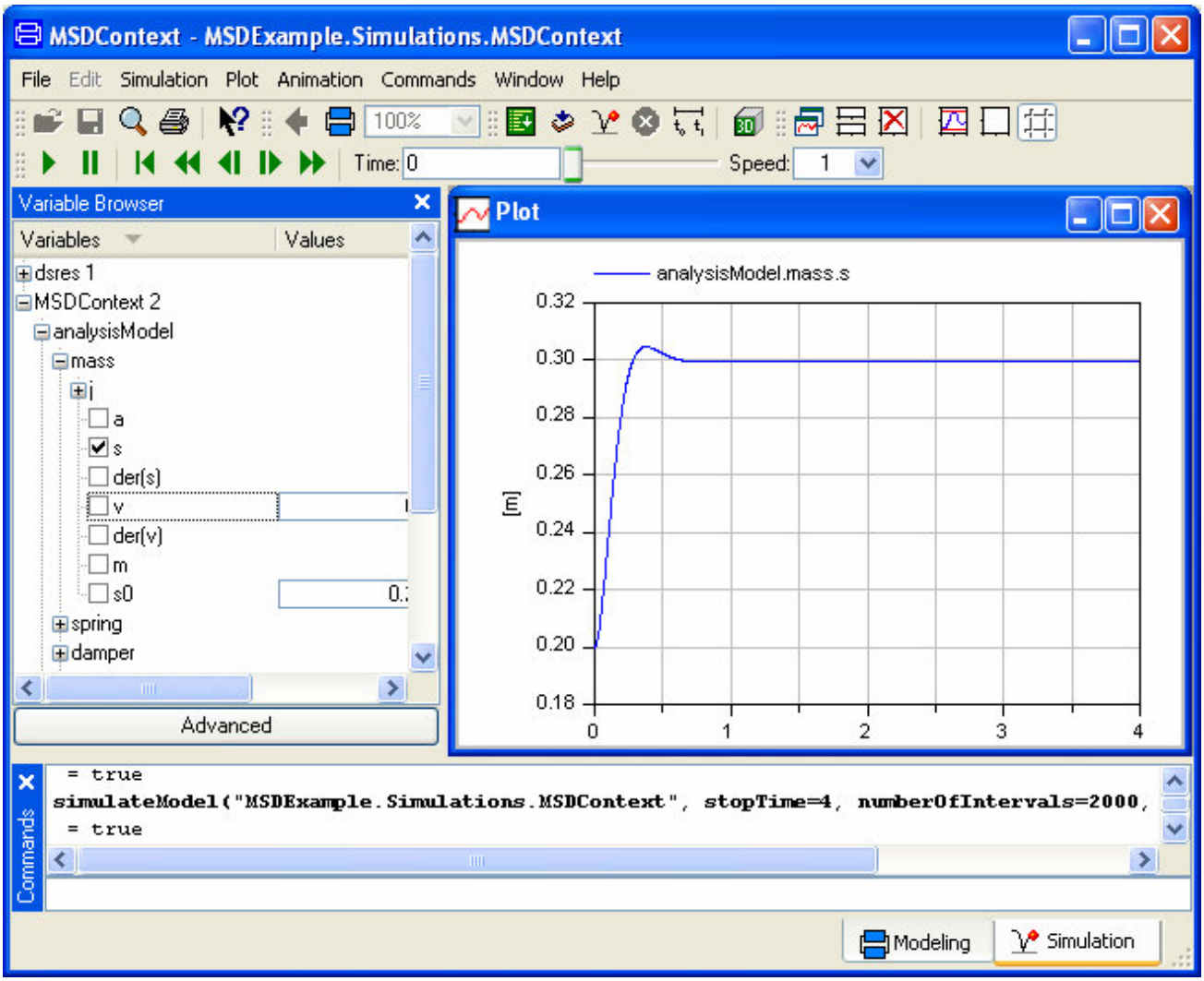

Figure 4.25: The Dymola simulation of the Modelica MSD model.

The simulation results of Figure 4.25 indicate that the MSD CD model authored in E+ was transformed into a meaningful, executable Modelica model. In fact, the performance of the simulated MSD system satisfies the *ReboundReq* requirement specified for the *WheelSuspension* modeled in Figure 2.1. Moreover, the behavior of the MSD model corresponds nicely with that of a true car suspension. When a suspension is given a displacement and forced to respond, it typically overshoots its steady-state position and gradually (i.e. with no residual vibration) settles.

While transforming a SysML CD model into Modelica provides some benefits for automating the simulation of the SysML model, the transformation of a model context (Sections 5.1 and 6.4) provides much more functionality for a SysML user. Transforming the model context enables the simulation of a CD model that includes information regarding static or known aspects of the system of interest. Currently, an unstable version of the SysMLTransformers plugin can handle some depictions of a CD model's context. This could be easily stabilized by continuing the development of graph transformation rules and the Java code used to run the transformations; however, the current abilities of the SysMLTransformers plugin provide promising examples for creating other types of graph transformations in support of model integration in SysML.

4.5 Summary

In this chapter, a TGG and operational graph transformational rules are presented to handle SysML-to-Modelica model transformations. Section 4.1 first justifies the selection of graph transformations for automating SysML-to-Modelica model transformations. Section 4.2 is a description of the SysML-Modelica TGG and the SysML-to-Modelica operational graph transformation rules. Section 4.3 is a discussion on the implementation of the SysML-Modelica TGG and graph transformation rules in the VIATRA graph transformation tool. Section 4.4 provides an overview of the SysMLTransformers plugin for RSD which is used to transform E+ SysML CD models into lexical Modelica models.

By establishing the ability to transform SysML models into Modelica models via graph transformations, a precedent has been set for enabling the execution of more complex graph transformations. The TGG proposed in this chapter provides a foundation

that can be reused or extended to support model transformations like Modelica-to-SysML transformations and model synchronization transformations. Moreover, this chapter provides a basic guide for creating a true link between SysML and other external models via graph transformations. As support grows for creating transformation links between various types of integrated models, engineers will be better able to ensure information consistency and model traceability throughout the model-based design of a complex system.

CHAPTER 5

MODELING SIMULATIONS AND ANALYSES IN SYSML

In the context of model-based systems engineering, models and simulations allow systems engineers to investigate and predict the behavior of system alternatives without the need for physical prototyping. For example, a CD model of a MSD system can be used to simulate and predict the behavior of a car suspension alternative. This chapter describes how to relate a CD model to other relevant design information/knowledge in SysML by binding of model parameters in a *model context*; defining an experiment performed on a model in a *simulation*; defining a measure of effectiveness as the result of a simulation; and using an *abstracted simulation* in the context of system analysis. This complements the model transformation approach outlined in Chapter 4 and the model integration effort in general because it enables the transformation and execution of CD models that incorporate information from other SysML models.

5.1 Defining the Model Context

In systems engineering, a continuous dynamics model is always used in a particular context. Within this model context the elements of a system's structural model are bound parametrically to the corresponding elements of the analysis model. For example, when analyzing a set of car suspension alternatives, engineers can assume that the mass used in a MSD CD model is always one quarter of the car's mass even though the suspension characteristics vary amongst the alternatives.

In current practice, engineers do not always distinguish between the physical structure or system topology and the corresponding system behavior. For instance, it is

common practice to use an electric circuit diagram as the representation for defining both the circuit topology as well as the behavior of the circuit in a SPICE simulation [48]. As systems become more complex engineers often need to represent a system with multiple simulation models corresponding to different levels of abstraction or different disciplinary perspectives. The use of an explicit model context as suggested here facilitates the preservation of consistency amongst all the separate models. A similar approach to setting the context for an analysis model is demonstrated with the MRA CBAM concept [21].

To relate the structure to the behavior, a *model context* block is defined with two part properties: one usage of the system model and one usage of the analysis model. If mathematical relationships beyond simple equivalence exist between the known elements of the system model and the corresponding elements of the analysis model, additional constraint blocks can also be defined. Finally, a parametric diagram of the model context block is created to bind the known system elements to the corresponding analysis elements.

In the lower portion of Figure 5.1, the block *ModelContext* is defined as owning usages of *MSD*, *Car*, and a constraint block named *MassRelation*.

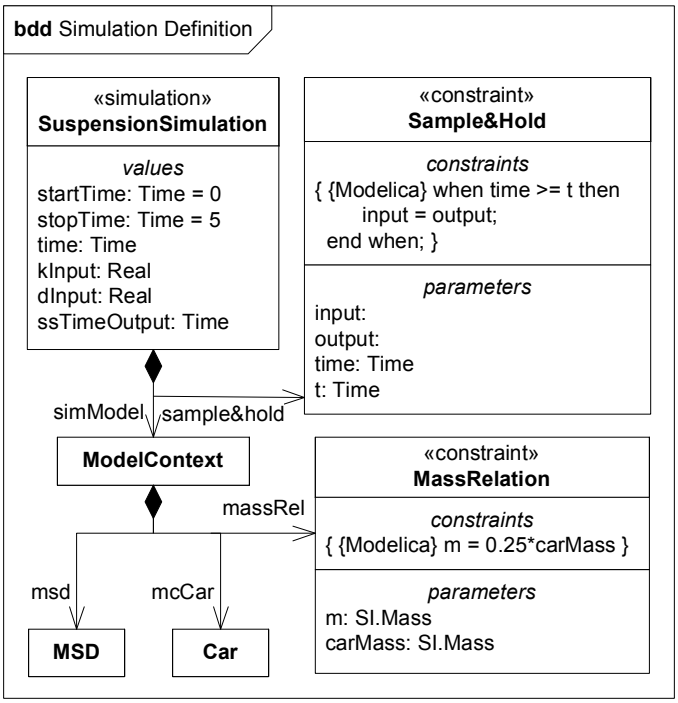

Figure 5.1: Declaration of the *SuspensionSimulation* and *ModelContext* blocks.

In Figure 5.2, a parametric diagram of *ModelContext* is used to establish the relationship between the masses of the MSD and car models.

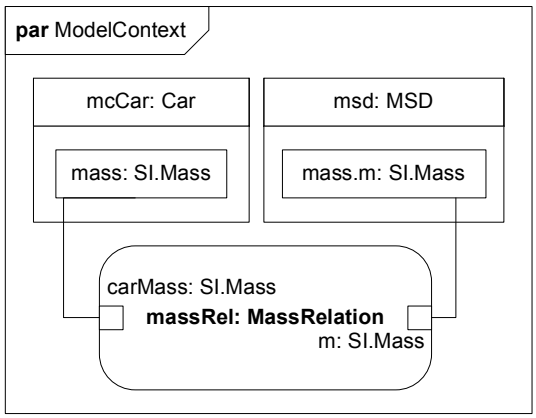

Figure 5.2: The parametric diagram of *ModelContext*.

Inside of this parametric diagram, *msd.mass.m* is defined as one quarter of the mass of *mcCar.mass* by connecting them to the appropriate parameters on the constraint property *massRel*.

5.2 Modeling the Simulation

A simulation is an experiment performed on a computational model [41]. Before a simulation can be performed, the experiment must be fully defined. A fully defined simulation includes a specification of initial conditions, boundary values, observed outputs, and potentially the process steps one must follow to complete the experiment. From a modeling perspective, all of these aspects can be captured in the computational model itself or in extensions of the model defined using the same modeling constructs described in Chapter 3. One can therefore assume that the "model" as defined in the model context is fully specified—all the parameters are bound to values and the set of system equations is non-singular. Under those assumptions, the only additional information that needs to be provided is the start and end time of the simulation.

To make the semantics of a simulation explicit in SysML, modelers can utilize the *«simulation»* stereotype. This original stereotype can be applied to a block that represents a simulation of a fully specified computational model. As is illustrated in Figure 5.1, this stereotype requires the inclusion of a *time* property, which represents the simulation time; *startTime* and *stopTime* properties; and a part property (e.g. *simModel*) that represents the computational model to be simulated. The semantics of the *«simulation»* stereotype are that all the properties in the computational model are evaluated as a function of *time* from *startTime* to *stopTime*. Note that the application of this stereotype completely defines a simulation experiment in a fashion that is independent of any particular simulation solver; however, other solver-specific properties could be included (e.g. number of intervals). In addition, note that Modelica semantics

differ from SysML semantics which require the explicit definition of a local simulation time property to which all time-varying system properties can be bound.

5.3 Abstracting the Simulation

A simulation as defined in the previous section allows a systems engineer to define an experiment in which the system behavior can be observed. However, simulations are often used to make system-level design decisions. In that case, the same experiment is often performed on multiple system alternatives. It then becomes important to abstract this simulation formally for reuse purposes by clearly defining the inputs (the properties that can take on different values from one simulation run to the next), and the outputs (the properties that are of interest to a decision maker, for instance, a measure of effectiveness that drives a design optimization). The relationship between simulation inputs and outputs can then itself be considered as a model. Unlike the model of the system, this model is an algebraic relationship, albeit a very complex one, that requires running the entire simulation to compute the outputs from the inputs. When abstracting a simulation in this fashion to support decision making, it is justifiable to assume that the outputs of the simulation are scalar quantities (decisions can only be made based on scalars because vectors cannot be rank-ordered [49]). Sometimes this requires that a modeler include additional modeling elements in the CD model to define these scalar measures of effectiveness. For instance, in the BDD in Figure 5.1 and the corresponding parametric diagram in Figure 5.3, the suspension simulation has been abstracted into an input-output model with inputs as the decision variables, *dInput* and *kInput* (bound to the damping and stiffness of the suspension), and an output as the measure of effectiveness, *ssTimeOutput* (the time to steady-state for the MSD system).

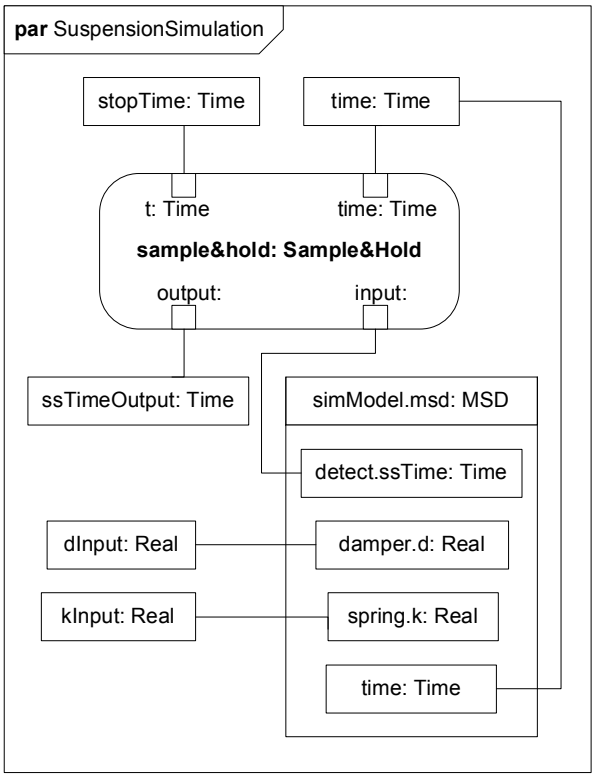

Figure 5.3: The parametric diagram of *SuspensionSimulation*.

The output has been bound to a property of MSD through a "sample and hold" constraint property, *sample&hold*, making explicit that the output takes on the value of the time-varying property *detect.ssTime* when the simulation time equals *stopTime*. In general, more complex models may be necessary to relate scalar outputs to time-varying simulation properties.

5.4 Embedding the Simulation into an Analysis

Once a simulation has been abstracted into an input-output model, it can be used in support of analyzing system alternatives with respect to stakeholder requirements and measures of effectiveness, as is illustrated in Figure 5.4 and Figure 5.5.

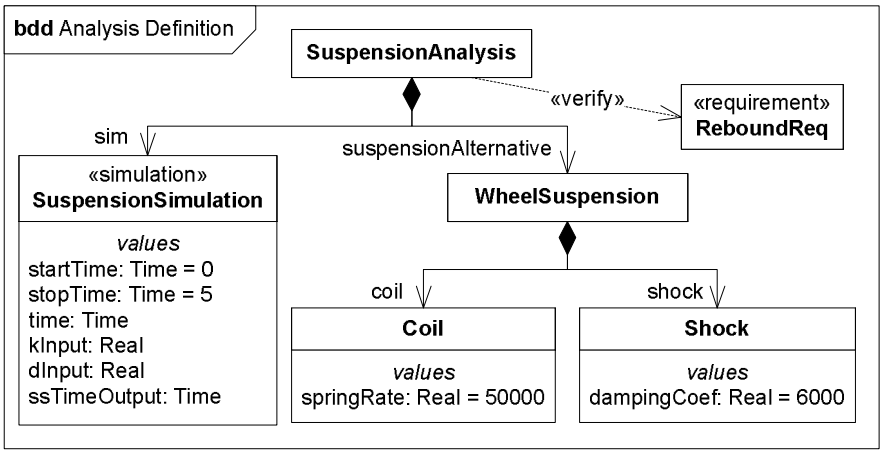

Figure 5.4: Declaration of the *SuspensionAnalysis* block.

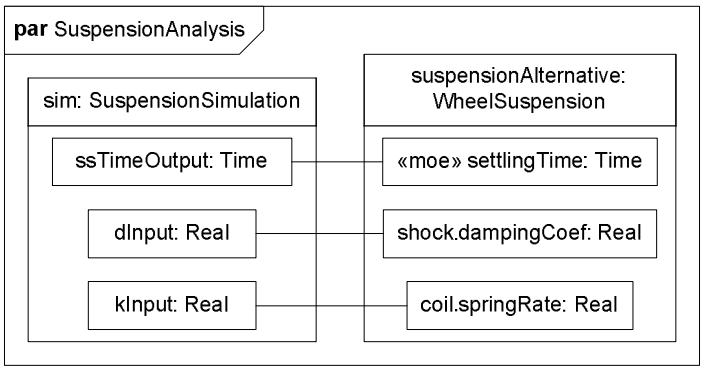

Figure 5.5: The parametric diagram of *SuspensionAnalysis*.

Analyses generally verify that a system alternative meets a certain system requirement. This can be modeled explicitly in SysML using the *«verify»* dependency. A parametric diagram of a block representing a system analysis can be used to connect the system alternative to the abstracted simulation, as illustrated in Figure 5.5. Instead of binding the simulation inputs and outputs directly to the corresponding value properties of the system alternative, one could also define an optimization problem in which the stiffness and damping are optimized with respect to one or more measures of effectiveness. Whenever there is a need for repeated evaluation of the simulation with different inputs, it is desirable to embed the simulation explicitly in an analysis as depicted in Figure 5.4.

5.5 Summary

This chapter presents the final facet of integrating a CD model into SysML by describing an approach to relating a SysML CD model to other elements of a SysML model via the creation of SysML models of simulations and engineering analyses. Section 5.1 is a description of how to set the context of a CD model by binding its properties to the properties of a SysML structural model. Section 5.2 is an explanation on depicting simulations of SysML CD models using common SysML modeling constructs. Section 5.3 describes the abstraction of simulation models for the purpose of enabling simulation reusability. Section 5.4 then discusses the creation of SysML models of engineering analyses that rely on abstracted simulation and system alternative models.

While others may approach the implementation of depicting system analyses differently than the approach outlined in this chapter, the basic concepts of modeling simulations and analyses in SysML are crucial for establishing meaningful relationships between CD and other SysML models. An analysis model like a SysML CD model provides little value to an engineer if it cannot answer a question about the system through simulation; hence, simulations and their owning analyses are a primary means of relating the knowledge contained in CD models and the knowledge contained in other design and analysis models.

By enabling the relation of CD models to other SysML models (e.g. structural model of a system alternative), the prospect of using model transformations as described in Chapter 4 becomes even more promising. Transforming a SysML CD model whose properties are bound to the properties of other SysML design or analysis models supplies

an executable Modelica model with information that sets the context for simulating the continuous behavior of a given system alternative.

CHAPTER 6

THE HYDRAULICALLY POWERED EXCAVATOR MODEL

6.1 Introduction to the Excavator Example

The example model presented in this chapter is intended to demonstrate the scalability of the CD model integration approach proposed in Chapter 3 through Chapter 5. If the approach is capable of handling the integration of complex models such as the excavator model, then its use in a MBSE design process could benefit engineers designing complex systems.

6.1.1 Overview of the Excavator Example

The model described in this chapter is meant to depict the continuous dynamic behavior of an earth-moving, hydraulically powered excavator. These machines are used extensively in the construction industry amongst others for performing a large variety of tasks with the most common being digging and trenching. They are complex systems composed of numerous interconnected subsystems and components and are typically designed by large companies employing distributed services from engineers of multiple disciplines.

Motion is provided to these systems through the complex control of multiple hydraulic actuators linked to various mechanical structures like the driver's carriage and the digging arm. The carriage is allowed to rotate about its base through the use of a hydraulic motor. The arm is composed of three main structures: the boom (the large mechanical link connected to the carriage), the crowd (the smaller mechanical link

between the boom and the bucket), and the digging bucket attached at the end of the excavator arm. The arm is allowed to move in three degrees of freedom through the use of four double-acting hydraulic cylinders: two parallel cylinders controlling the boom rotation, one controlling the crowd rotation, and one controlling the bucket rotation. The hydraulic actuators are powered by a load-sensing, pressure-compensating circuit controlling the operation of a variable-displacement hydraulic pump. The pump is typically driven by an internal-combustion engine. Flow is routed to the actuators through the use of four load-sensing directional servo valves. The valve positions are continuously controlled by an excavator operator through control signals typically input from a joystick interface.

To model the digging motion of a hydraulically powered excavator, a Modelica model can contain an enormous set of hybrid discrete-event and DAE models. Both the SysML and Modelica excavator CD models depicted in this chapter represent a collection of over 11,000 equations. The CD model primarily captures the energy-based, continuous behavior of the rigid-body mechanics and the hydraulics, but also includes simplified models of the control signals and the environment.

6.1.2 Appropriateness of the Example Model

This model was chosen to test the abilities of the model integration approach outlined in this thesis due to its increased complexity and relevance to the systems engineering community as compared to the simple car suspension model discussed in Chapter 2 through Chapter 5. The excavator model is complex due to its multiple degrees of freedom, subsystems, and encompassed engineering disciplines. Such a model if deemed valid can provide a large amount of valuable information for a decision maker

selecting or eliminating individual alternatives from large discrete or continuous design spaces.

Under the assumption that the model is sufficiently complex for testing the abilities of the SysML CD model integration approach, the rest of this chapter utilizes the principles of the approach to integrate the excavator CD model into SysML via its depiction using SysML modeling constructs, the transformation of the SysML CD model into Modelica code, and the incorporation of the SysML CD model into a simulation and engineering analysis model. The model is developed using the E+ toolkit for RSD which imposes certain modeling limitations with respect to the integration approach proposed in this thesis. These limitations are identified throughout the description of the E+ SysML CD model.

6.2 Defining the SysML CD Model of the Excavator

To begin the integration process, the excavator CD model is first declared and composed using the "white box" and "black box" approaches outlined in Sections 3.3 and 3.4, respectively. First, as seen in Figure 6.1, an original SysML block, *ExcavatorDigCycle*, is declared in a BDD as a CD model of the excavator's dig cycle.

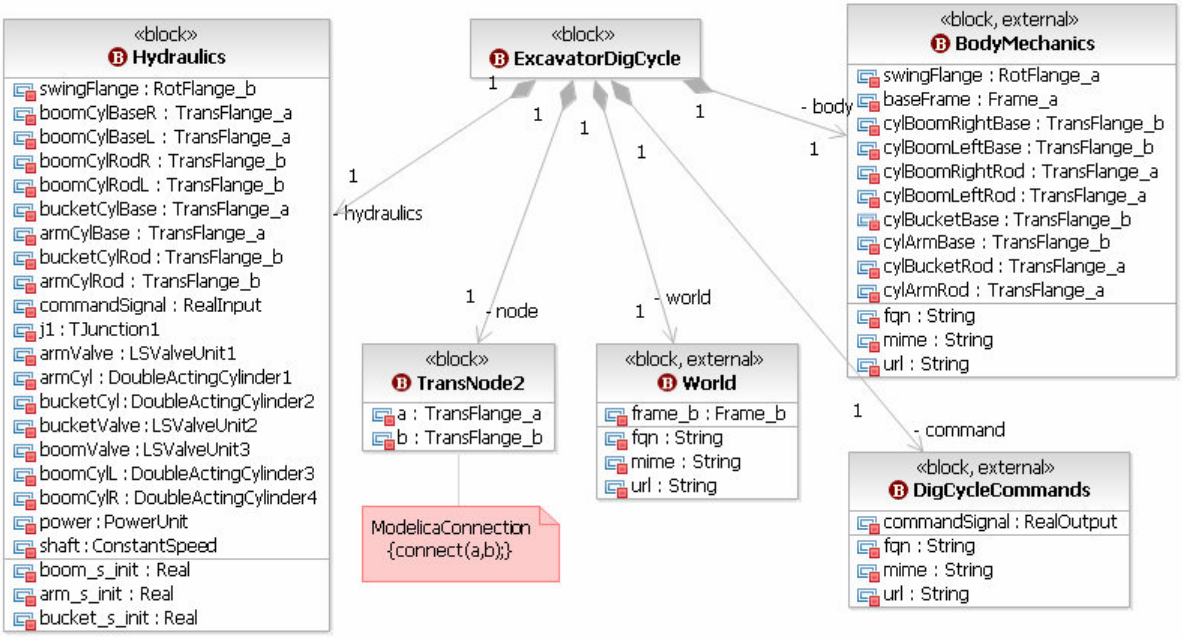

Figure 6.1: The BDD of the *ExcavatorDigCycle* SysML CD model.

The block *ExcavatorDigCycle* is decomposed into various part properties typed to other *blocks*: three external, "black box" blocks representing pre-existing Modelica models of the excavator's multi-body mechanical structure, dig-cycle command signals, and a world reference frame; one original "white box" block representing the hydraulics subsystem; and one block representing a system node for demonstrating the equivalence of Modelica-specific system nodes and the *«connectClause»* binding connector.

At this point, it is necessary to discuss an E+ limitation affecting the depiction of system nodes and constraint blocks in general. While the modeling approach outlined in Sections 3.3.5 and 3.4.3 promotes the use of constraint blocks for depicting system nodes, bugs in the E+ toolkit prevent a user from following the approach exactly. More specifically, a constraint parameter typed to a block instead of a value or data type cannot be connected to any other elements using assembly or binding connectors. This means that a user cannot connect a component interface typed to a block (e.g. a part property typed to *MechanicalJunction*) to a usage of a node constraint block since the constraint

parameter must be typed to the same block (e.g. a parameter typed to *MechanicalJunction*). To overcome this issue, a modeler must represent nodes using regular blocks instead of constraint blocks, as seen in Figure 6.1.

The hydraulics subsystem, modeled by *Hydraulics* in Figure 6.1, is further depicted in its BDD seen in Figure 6.2.

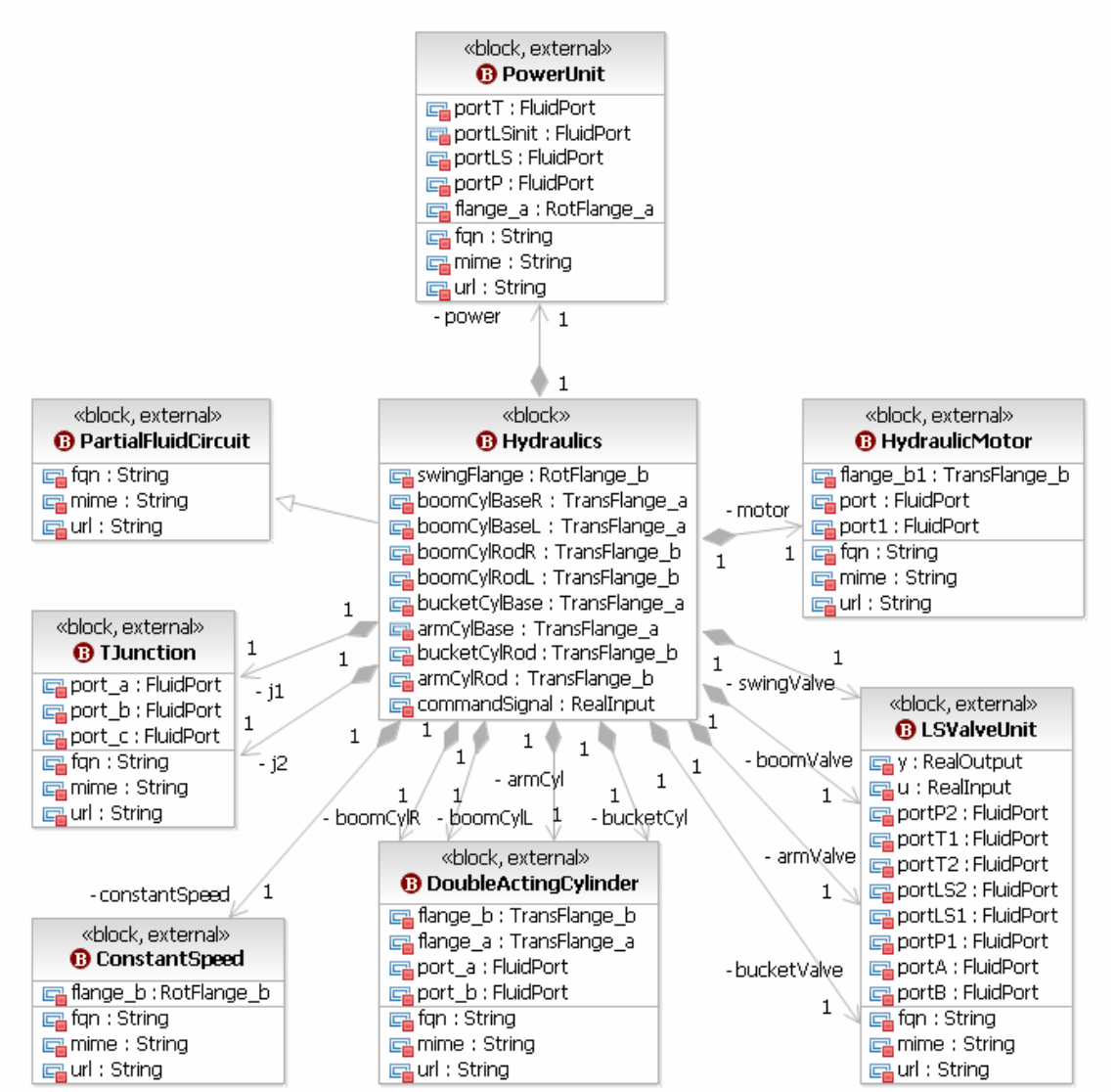

Figure 6.2: The BDD of the *Hydraulics* SysML CD sub-model.

Hydraulics is broken down into part properties representing its interface with the other excavator subsystem models and other properties representing the hydraulic components.

More specifically, the hydraulics subsystem is composed of relations to six different external blocks: *PartialFluidCircuit*, *TJunction*, *DoubleActingCylinder*, *LSValveUnit*, and *HydraulicMotor* from the FluidPower [50] library for Modelica; and the *ConstantSpeed* rotational-mechanical model from the MSL.

To compose the system CD model of the excavator, the multiple subsystems and components must be bound together using the approaches outlined in Sections 3.3.5 and 3.4.3. First the high-level *ExcavatorDigCycle* model is composed in an Internal Block Diagram (IBD) in place of a parametric diagram due to another E+ modeling limitation. When modeling in an E+ parametric diagram, a binding connector isn't owned by the diagram owner if the connector is placed between nested properties belonging to two different part properties. Instead, the connector is incorrectly owned by the definition block of one of the part properties. For example, if a connector is drawn between *A.b.c* and *A.d.c* in a parametric diagram of *A* while *b* is typed to *B* and *d* is typed to *D*, the connector is incorrectly placed between *B.c* and *D.c* and owned by either *B* or *D*; however, this is not the case when modeling in an IBD. When composing a system model in an IBD, nested connector ends are correctly placed between nested properties. To cope with this problem, system models are composed in E+ IBDs instead of parametric diagrams. The IBD of *ExcavatoDigCycle* is illustrated in Figure 6.3.

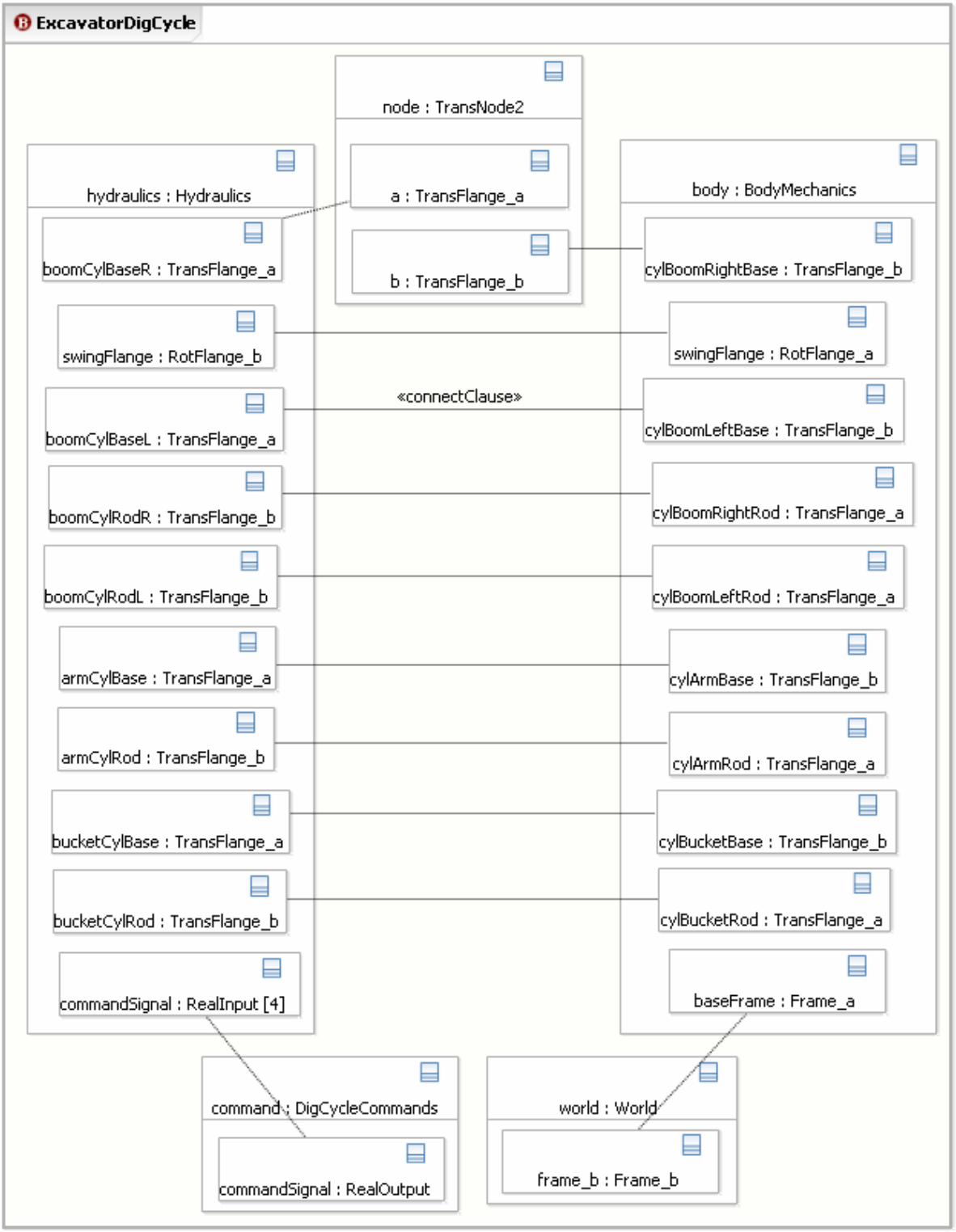

Figure 6.3: The IBD of *ExcavatorDicCycle*.

Figure 6.3 depicts the equivalence between Modelica-specific system nodes and *«connectClause»* binding connectors. Any connectors bypassing a system node are assumed to have the *«connectClause»* stereotype. Figure 6.4 displays a much larger and more complex model composition through the depiction of *Hydraulic*'s IBD.

Figure 6.4 also depicts a work-around for overcoming another E+ bug. When making connections between multiple usages of the same block in an IBD, connectors are often incorrectly and automatically placed in other parts of the same diagram. Suppose that block *A* has two properties *b* and *c*. If another block *D* owns two usages of *A*, *a1* and *a2*, and a connector is drawn from *a1.b* to another property in *D*, say *e.f*, another connector automatically appears in the IBD of *D* between *a2.b* and *e.f*. To overcome this problem, every E+ definition block can only be typed by one property in a given block. If a definition block is required for two or more properties of one block, it is copied and renamed as many times as necessary. For instance, instead of creating four usages of *DoubleActingCylinder* in *Hydraulics* (as seen in Figure 6.2), four part properties are typed to four independent definition blocks containing the same definition: *DoubleActingCylinder1*, *DoubleActingCylinder2*, *DoubleActingCylinder3*, and *DoubleActingCylinder4*.

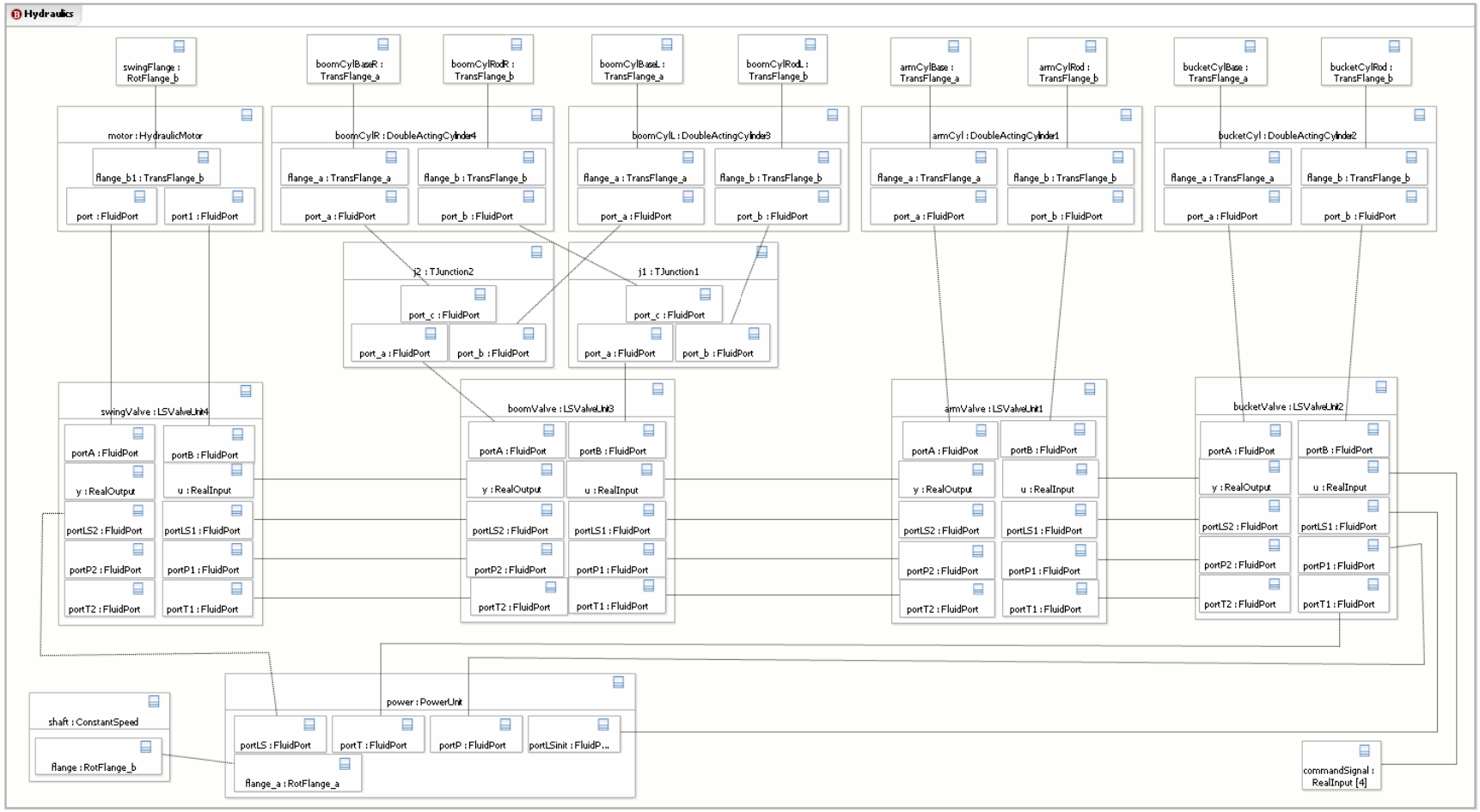

Figure 6.4: The IBD of *Hydraulics*.

This intricate IBD demonstrates the ability of the CD modeling approach to capture the behavior of complex engineered systems. This final diagram concludes the depiction of the excavator CD model which is now ready to be transformed into a corresponding Modelica model. One must note that this intricate SysML model was not the only way to integrate an excavator CD model into SysML. Instead, a modeler could have modeled the entire excavator model in Modelica and referred to it using an external block.

6.3 Transforming the SysML Excavator Model

This section builds upon the work presented in Section 6.2 by transforming the SysML CD model of the excavator into an executable Modelica model. Just as the MSD model was transformed in Section 4.4, the SysMLTransformers plugin for RSD/E+ is used to transform the excavator SysML CD model using the SysML-Modelica TGG and operational graph transformation rules implemented in VIATRA. An excerpt of the resulting Modelica model as displayed in MDT can be seen in Figure 6.5.

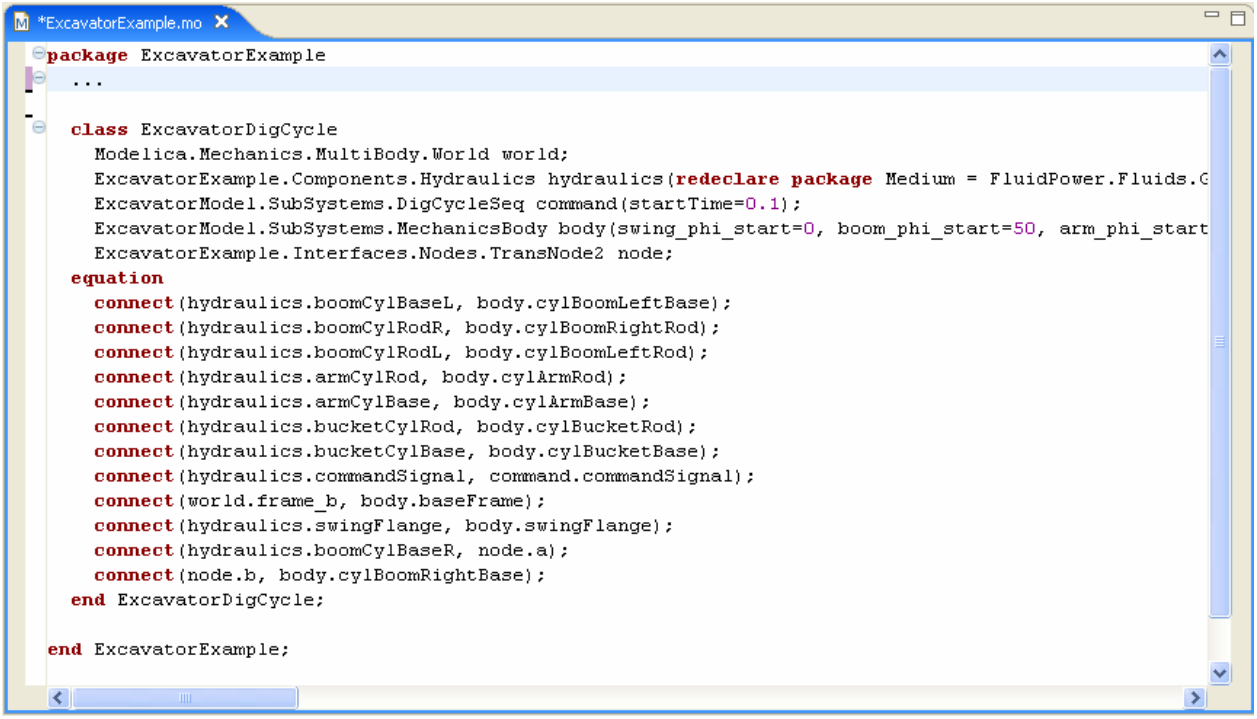

Figure 6.5: An MDT view of the Modelica *ExcavatorExample* model.

This Modelica model can then be imported into Dymola for execution. This is illustrated in Figure 6.6.

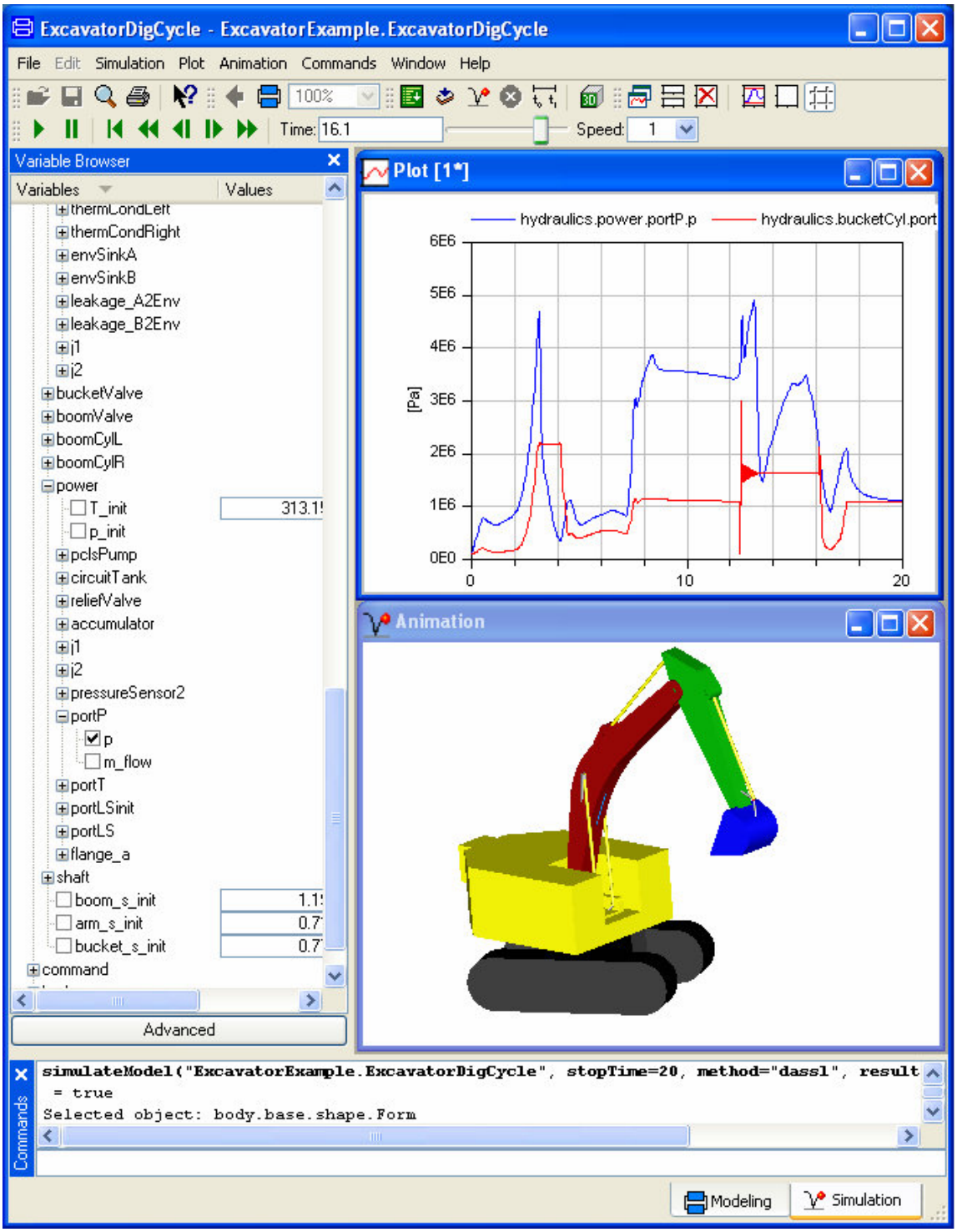

Figure 6.6: A Dymola simulation and animation of the *ExcavatorDigCycle* model.

The results of the SysML-based Dymola simulation seen in Figure 6.6 correspond with the results obtained by manually building the *same* excavator model directly in Modelica

syntax (thus validating that the new auto-generated approach produces the same model as the traditional, manual method). In fact, the Modelica excavator model has been under manual, iterative development for over a year and provides meaningful results with respect to the actual behavior of a hydraulically powered excavator. This is encouraging because the SysML representation is appropriately abstracting the behavior of a complex model that has been painstakingly developed in support of testing the open source Fluid Power Modelica library [50]. However, work still needs to be done on the model. Aside from adding more detail (if higher fidelity results are desired), one can see from Figure 6.6 that the damping of the system should increase to combat the pressure fluctuations seen in port A of the bucket cylinder past a time of 12 seconds during the dig cycle.

6.4 Integrating the Excavator Model into a Simulation and Analysis

The final step in completing the SysML integration of the excavator CD model is the establishment of its relationships with other elements of the larger SysML information model via models of a dig cycle simulation and corresponding engineering analysis. First, Figure 6.7 and Figure 6.8 set the context for the excavator CD model by binding one of its properties to the *mass* property of a *Carriage* structural model.

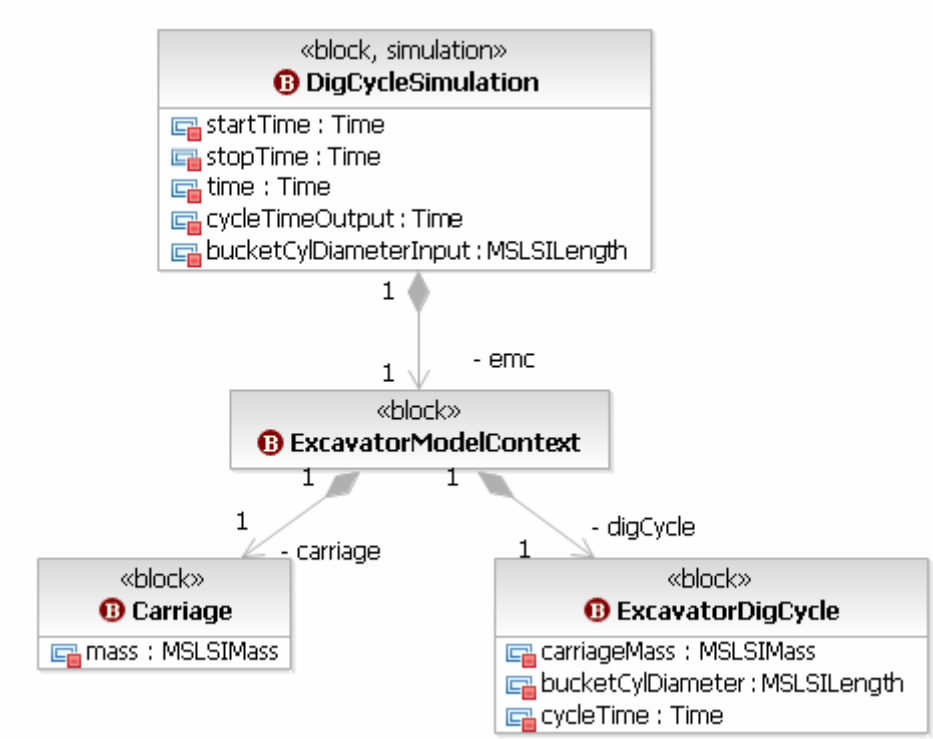

Figure 6.7: The BDD of *DigCycleSimulation* and *ExcavatorModelContext*.

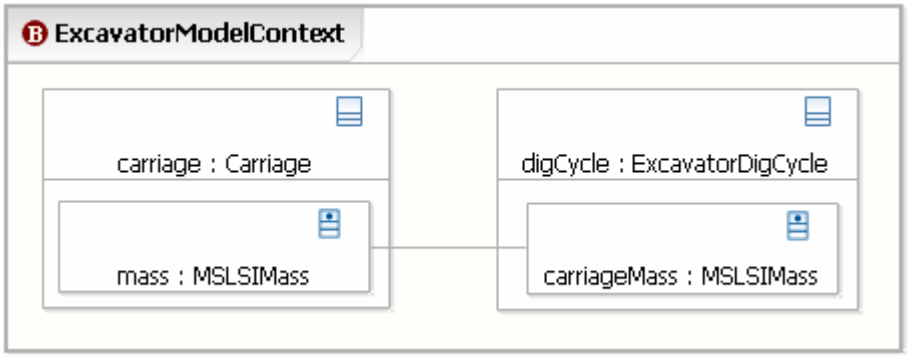

Figure 6.8: The IBD of *ExcavatorModelContext*.

Figure 6.7 also depicts the definition of a SysML simulation block named *DigCycleSimulation* which is assigned the *«simulation»* stereotype and the accompanying *startTime*, *stopTime*, and *time* value properties. The simulation model is abstracted into a reusable input-output model, as seen in Figure 6.7 and Figure 6.9, by assigning it the

values *bucketCylDiameterInput* and *cycleTimeOutput* and binding them to the corresponding properties of *emc.digCycle*: *ExcavatorDigCycle*

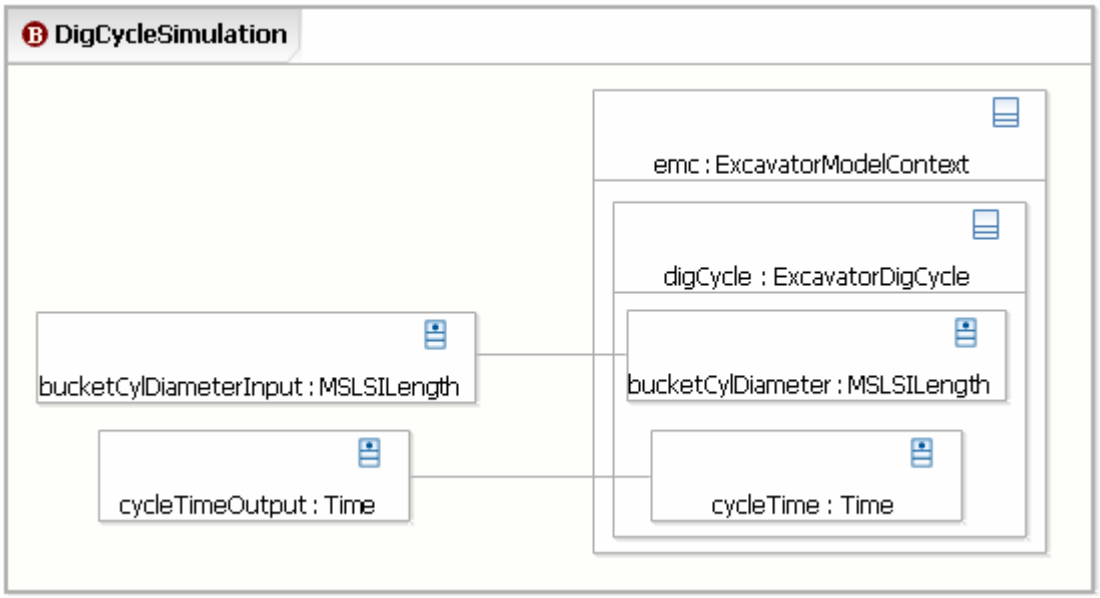

Figure 6.9: The simulation abstraction IBD of *DigCycleSimulation*.

Finally, the integration of the excavator CD model is completed by embedding the abstracted simulation model into a model of an engineering analysis of a system alternative model. This is illustrated in Figure 6.10 and Figure 6.11.

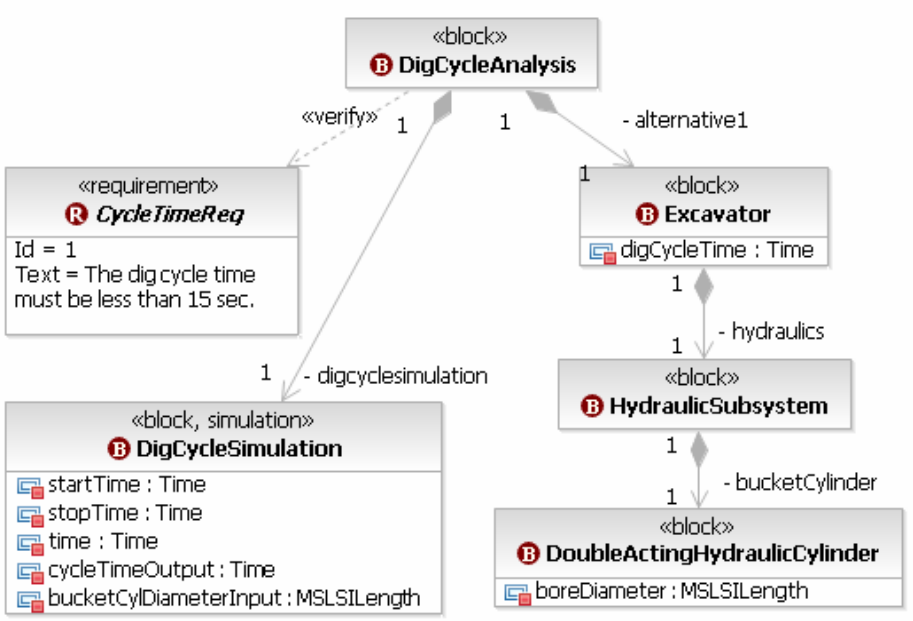

Figure 6.10: The BDD of *DigCycleAnalysis*.

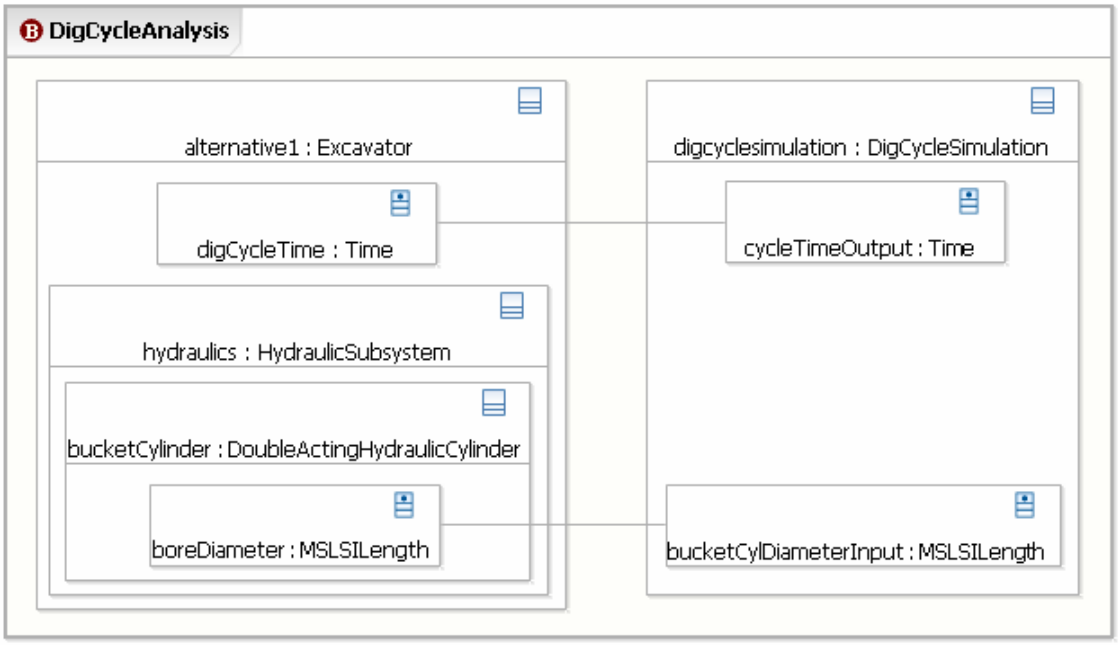

Figure 6.11: The IBD of *DigCycleAnalysis*.

6.5 Summary

The intent of this chapter is provide an example of integrating into SysML a CD model that goes far beyond the complexity of the MSD example initially presented in this

thesis. Section 6.1 provides a brief introduction to the excavator example and a justification of its use in this thesis. Section 6.2 begins the description of the excavator example by declaring the *ExcavatorDigCycle* SysML CD model. Section 6.3 then demonstrates the use of this SysML CD model for automatically generating a corresponding, executable Modelica CD model. Finally, Section 6.4 completes the model integration process by relating the excavator CD model to other elements in the SysML information model through the creation of models representing the dig cycle simulation and a corresponding dig cycle analysis.

CHAPTER 7

DISCUSSION AND CLOSURE

In this thesis, CD models representing continuous dynamic system behavior are integrated into SysML to further promote and support a shift to MBSE for complex systems design. This final chapter discusses the integration abilities contributed in this thesis by discussing their validity, limitations, and future prospects. The thesis is then brought to a close with some final remarks.

7.1 Review and Evaluation of the Model Integration Approach

The driver behind this thesis is an open-ended question about the use of design and analysis model integration via SysML for the promotion of information consistency, model traceability, and automated model transformation. Many people have explored model integration in SysML (e.g. Peak et al. [20], Hooman et al. [29], Huang et al. [24]), but this thesis specifically focuses on the use of a language mapping; TGG and graph transformation rules; and models of simulations and engineering analyses to support the integration of Modelica representations of CD into SysML information models. Consequently, a "model" of sorts is provided for integrating CD models and, if the "model" is sufficiently generalized, other design and analysis models into larger SysML models.

Whenever an engineer decides to use a model, he/she must ensure that the model is valid with respect to the conditions under which the model is used. Hence, if distributed engineers developing complex systems are to use or extend the model

integration approach outlined in this thesis, they must be sure that the method is valid for their purposes.

To verify and validate methods and models related to engineering design, one tool that is commonly utilized is the validation square [51]. Due to high level of relevance between the work presented in this thesis and the field of engineering design, the validation square is used to evaluate the model integration approach. The validation square, as seen in Figure 7.1, is decomposed into four quadrants representing the necessary validation steps.

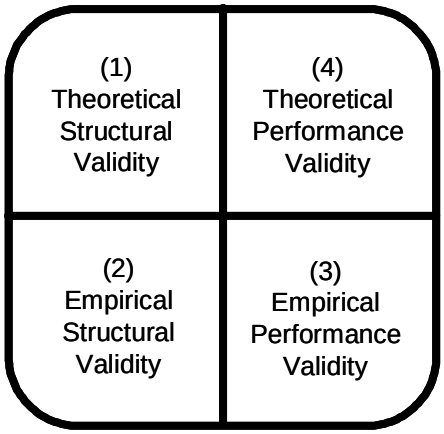

Figure 7.1: The validation square [51].

To validate some construct or piece of work, a user must first ensure that the construct has theoretical structural validity. This requires the user to ensure that the construct is logically consistent. When a user can confidently make that assertion, he/she can move onto ensuring empirical structural validity. In this quadrant, the user must build confidence in the example problems used to test the construct. If the user is confident in his choice of example problems, he/she can move onto empirical performance validity. During this phase of validation, the user has applied the construct to the example problems and is using the results as supporting evidence. The user must accept that the

example problems provide meaningful results. Upon satisfying this quadrant, the user can move onto theoretical performance validity. In this quadrant, the user must take a "leap of faith" by accepting that the construct is useful beyond the applications presented in the example problems. If this quadrant is satisfied, the construct has been validated and is generally applicable to the problems it was intended to solve.

To ensure that the integration approach maintains theoretical structural validity, the approach must be logically consistent and adept at integrating CD models into SysML to promote consistency, traceability, and automation. The steps used to integrate CD models into SysML enable the establishment of consistency links amongst the sub-models existing in the SysML information model. The integration approach also promotes traceability by enabling the establishment of dependencies and associations between various types of SysML models (e.g. requirements models) and models of simulations and engineering analyses which incorporate SysML CD models. Furthermore, the approach promotes automation by enabling the implementation of graph transformations for automatically transforming information/knowledge between SysML and Modelica models. Since the approach promotes consistency, traceability, and automation *and* is consistent with the motivation described in Section 1.4, the approach is deemed to be theoretically structurally valid.

To ensure empirical structural validity, confidence in the example problems (i.e. the MSD and excavator models) must be established. While the MSD example CD model is convenient for easily displaying the important aspects of the integration approach, it is not representative of the complex models encountered in contemporary systems engineering problems. On the other hand, the CD model of the hydraulically

powered excavator is certainly comparable to the complexity of contemporary systems, as argued in Section 6.1.2. When combined together, the MSD and excavator example models demonstrate the applicability of this model integration approach to problems of varying complexity. Hence, the work presented in this thesis is empirically structurally valid.

The empirical performance validity of the work presented is ascertained through the successful illustration of both the MSD and excavator example problems. When the integration approach is applied to both CD model examples, the result is an interconnected set of SysML constructs infusing an external CD model into a larger MBSE problem. These integrated models now promote consistency, traceability, and transformation automation in a way that better enables engineers to apply MBSE in the design of complex systems. Hence, empirical performance validity is established for the approach to integrating CD models into SysML.

To fulfill the last quadrant of the validation square, theoretical performance validity must be ensured for the integration approach. In other words, the integration approach must be applicable to problems outside of the MSD and excavator examples. As mentioned before, both examples span a large range of complexity. One can assume that the range represents or is close to the complexities encountered in the design of contemporary systems. Moreover, the approach could be generalized and reapplied to the integration of other design and analysis models further expanding its application base. 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 distributed design teams. One can assume that through improvement of implementation details this

approach to CD model integration could be valuable for the target audience; however, that value has yet to be confirmed. This can only occur through extensive user testing and improved implementations of the integration approach. Hence, while the work appears to be applicable to its intended audience and scenarios, theoretical performance validity is not completely ensured.

7.2 Limitations

Language Inconsistencies

The most fundamental limitation of this work is that the integration approach is based on a language mapping that is subject to various inconsistencies between SysML and Modelica. The first notable inconsistency is that Modelica offers restricted classes built in to the language for component definition while SysML only relies on the block and value type for property definition. In the case of mapping Modelica connectors to a SysML definition construct, the graph transformation in Figure 4.7 provides a suitable work around; however, many other Modelica restricted class types are ignored in this thesis. Another inconsistency between the two languages is Modelica's use of the variability prefixes like *flow* and *parameter*. While these have no direct equivalents in SysML, SysML properties could be further extended with stereotypes to match the semantics associated with the various Modelica variability prefixes. This lack of variability prefixes in SysML also causes an inconsistency between the semantics of a SysML binding connector and a Modelica connect clause. This inconsistency was discussed at length and resolved in Section 3.4.3.

Incomplete TGG subgraphs

Another fundamental limitation of this work is the use of incomplete and simplified metamodels during the construction of the TGG. Both the SysML and Modelica metamodel graphs omit some elements of and make questionable assumptions about their respective languages in an attempt to balance accuracy and usability. These simplifications and assumptions will not support all possible SysML and Modelica models. In the SysML metamodel, only elements from Chapter 8 and Chapter 10 of the SysML specification [9] are included in the metamodel. A complete metamodel would include every modeling element from SysML (e.g. requirements, use cases, activities, state machines). In the Modelica metamodel, different types of special equations are not treated as individual modeling elements per the Modelica specification [11] and are simply lumped together in the *equation* entity. Additionally, special variability prefixes (e.g. *input*, *output*, *constant*, *final*) and restricted classes (e.g. *functions*, *records*, *models*) are ignored. Moreover, the correspondence metamodel maintains traceability between actual SysML and Modelica modeling entities but ignores the correspondence between various SysML and Modelica modeling relations. The majority of the extensions required to complete the TGG are implementation-oriented (i.e., they can be implemented using the same concepts described in this thesis); however, others may require conceptual extensions beyond the method described in this thesis. A complete TGG should relate *every* aspect (entity or relation) of one language to another.

Reliance on Modelica 2.2

The work presented in this thesis also has limitations from the implementation perspective. The first limitation is the dependence upon version 2.2 [11] of the Modelica

language specification. During the course of this work, version 3.0 [52] Modelica was released and some of its constructs are not supported by the language mapping and graph transformations. For instance, the concept of a *replaceable package* (i.e. a package that serves as a template package and can be later specialized through redefinition) has been added to Modelica 3.0, but has not been addressed in this thesis.

Focus on Operational Graph Transformation Rules

Another implementation-oriented limitation of this work is its focus on operational graph transformations for enabling the generation of Modelica CD models from SysML CD models. While these transformation abilities showcase the potential of using graph transformations for integrating SysML models with external models, they don't actually provide other necessary abilities. These include the ability to synchronize SysML and Modelica models and the ability to generate SysML models from Modelica code. Both of these abilities could be achieved through the creation of bidirectional transformation rules that force a SysML and Modelica to adhere to the TGG described in Section 4.2, but the development of such rules requires further development and an increased understanding of graph transformation theory.

Non-executable Models of Engineering Analyses

One last implementation-oriented limitation of this work is the current inability to execute SysML models of simulations and engineering analysis. Currently, simulations and model contexts can be handled by an unstable version of the SysMLTransformers plugin, but SysML models of engineering analyses are not handled. Such an ability is crucial for increasing the credibility and power of MBSE. Without this ability, the work

presented in Chapter 6 only enables systems engineers to design and document a simulation or engineering analysis.

Practical Limitations

With respect to the practicality of the integration approach, the work presented in this thesis is likely to only provide value to geographically distributed businesses designing complex systems. Until model integration is better supported with easy-to-use software tools, the added overhead of using advanced model integration in simpler design projects is likely to detract value during the design process. Another practical limitation of this work is that it has not been tested by its target audience. Moreover, performing such tests in conjunction with this work is not currently a feasible prospect. To test the utility of this work, large shifts would need to occur from document-centric design to MBSE in the systems engineering community. Only then would a sufficient user base exist for testing the approach to CD model integration.

7.3 Future Work

The direction of future work should first point towards the development of a more robust and comprehensive SysML-Modelica mapping via the TGG schema, better transformation rules, and a stable software tool that can be presented and tested in industry and academia. As mentioned in Section 4.4, the current implementation of the graph transformer is proficient at transforming a context-free SysML CD model, but not fully able to transform CD models wrapped into a model context. To ensure the success of SysML as a model integration platform, such functionality must be acquired to

increase support for information consistency, model traceability, and automated CD model transformation and execution.

Furthermore, consideration should be given to the integration of a powerful engineering analysis tool/language, like ModelCenter [53], for actually executing a SysML model of an engineering analysis composed of a heterogeneous set of smaller design and analysis models bound or belonging to abstracted simulations. First of all, such integration would enable system alternatives described in SysML to be analyzed automatically in ModelCenter based on multiple system aspects (e.g. structural, CD, cost). Such an accomplishment could push the boundaries of model integration and advance the current state of concurrent engineering practices.

To increase credibility in the claim that this approach can be generalized and re-specialized for integrating other design and analysis models into a SysML model, the general approach should be applied to engineering modeling languages commonly used in the development of complex systems. For instance, such languages include Maple [54], CAD modeling languages, and finite element languages. In a fashion similar to the approach outlined in this thesis, integration should be achieved through language mappings, graph transformation schemas, and the formal representation of simulations and engineering analyses.

7.4 Closing Remarks

As systems design becomes an increasingly complex endeavor, engineers must be able to manage effectively the large quantities of associated design information and knowledge. Moreover, as design teams continue to lose the sense of central locality, the use of document-centric design continues to become an antiquated and error-prone

approach to solving systems engineering problems. In contrast with document-centric design, MBSE encourages designers to accept and adapt to the changes permeating the field of systems engineering.

To improve support for MBSE, this thesis builds upon the notion that SysML is a platform for model integration by exploring the synergy between SysML and Modelica. By creating a language mapping between SysML and Modelica, an approach is provided for representing system CD models alongside other SysML models used to capture a systems engineering problem. Graph transformations are then utilized for creating execution links between SysML and Modelica to support model generation and synchronization. Finally, an approach is outlined for relating a CD model to other SysML models via the specification of simulations and engineering analyses.

Hopefully, the work in this thesis not only enables the integration of CD models, but also encourages and provides guidance for other researchers attempting to improve support for model integration and MBSE in general. To succeed in the competitive global marketplace, designers must be adaptable and forward-thinking. Clearly, the continued development and adoption of MBSE is a useful tactic for adapting to the changing times; however, MBSE is still a relatively young approach to systems design and requires continuous nurturing from industrial and academic champions. The work presented in this thesis is just one more stride towards realizing the wide-spread use of model integration and MBSE.

REFERENCES

[1] Sage, A. P., and Armstrong Jr., J. E., 2000, *Introduction to Systems Engineering*, John Wiley & Sons, Inc., New York, NY.

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

[3] Forsberg, K., and Mooz, H., 1992, "The Relationship of Systems Engineering to the Project Cycle," *Engineering Management Journal*, **4**(3), pp. 36-43.

[4] Oliver, D., Kelliher, T. P., and Keegan, Jr., J. G, 1997, *Engineering Complex Systems with Models and Objects*, McGraw-Hill, New York.

[5] Estefan, J., 2007, "Survey of Model-Based Systems Engineering (MBSE) Methodologies," Jet Propulsion Laboratory, California Institute of Technology, Pasedena, CA

[6] Gero, J. S., 1990, "Design Prototypes: A Knowledge Representation Schema for Design," *AI Magazine*, 11(4), pp. 26-36.

[7] Mylopoulos, J., 1998, "Information Modeling in the Time of the Revolution," *Information Systems*, **23**(3-4).

[8] ISO/IEC, 2005, "Unified Modeling Language Specification," http://www.omg.org/cgi-bin/apps/doc?formal/05-04-01.pdf, April 2008.

[9] Object Management Group, 2007, "OMG Systems Modeling Language Specification," http://www.omg.org/cgi-bin/doc?ptc/07-09-01, April 2008.

[10] Booch, G., Jacobson, I., and Rumbaugh, J., 2005, *The Unified Modeling Language User Guide*, Addison-Wesley Professional.

[11] Modelica Association, 2005, "Modelica Language Specification Version 2.2," http://www.modelica.org/documents/ModelicaSpec22.pdf, April 2008.

[12] The Mathworks, 2008, *Simulink*, http://www.mathworks.com/products/simulink/, April 2008.

[13] 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.

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

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

[16] 2006, "The VIATRA 2 Model Transformation Framework: User's Guide," http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/doc/viatratut_October2006.pdf, April 2008.

[17] Varró, D., 2003, *VIATRA: Visual Automated Model Transformation*, Thesis, Department of Measurement and Information Systems, University of Technology and Economics, Budapest.

[18] The Eclipse Foundation, 2008, *Eclipse*, http://www.eclipse.org/, April 2008.

[19] IBM, 2007, *Rational Systems Developer (RSD)*, http://www.ibm.com/developerworks/rational/products/rsd/, April 2008.

[20] Peak, R. S., Burkhart, R. M., Friedenthal, S. A., Wilson, M. W., Bajaj, M., and Kim, I., 2007, "Simulation-Based Design Using SysML—Part 1: A Parametrics Primer," *INCOSE International Symposium*, San Diego, CA.

[21] Peak, R. S., Burkhart, R. M., Friedenthal, S. A., Wilson, M. W., Bajaj, M., and Kim, I., 2007, "Simulation-Based Design Using SysML—Part 2: Celebrating Diversity by Example," *INCOSE International Symposium*, San Diego, CA.

[22] 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.

[23] 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.

[24] Huang, E., Ramamurthy, R., and McGinnis, L., 2007, "System and Simulation Modeling Using SysML," in *The 2007 Winter Simulation Conference*, Washington, D. C.

[25] Tecnomatix, 2003, *eM-Plant*, http://www.emplant.com, April 2008.

[26] W3C, 2007, "XML Path Language (XPath) Version 1.0," http://www.w3.org/TR/xpath, April 2008.

[27] Jobe, J. M., Johnson, T. A., and Paredis, C. J. J., 2008, "Multi-Aspect Component Models: A Framework for Model Reuse in SysML," in *ASME 2008 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE 2008)*, Brooklyn, NY.

[28] Vanderperren, Y., and Dehaene, W., 2006, "From UML/SysML to Matlab/Simulink: Current State and Future Perspectives," in *Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany.

[29] Hooman, J., Mulyar, N., and Posta, L., 2004, "Coupling Simulink and UML Models," in *Symposium FORMS/FORMATS*.

[30] Telelogic, 2008, *Rhapsody*, http://modeling.telelogic.com/products/rhapsody/index.cfm, April 2008.

[31] Reichmann, C., Gebauer, D., and Müller-Glaser, K. D., 2004, "Model Level Coupling of Hetergeneous Embedded Systems," in *2nd RTAS Workshop on Model-Driven Embedded Systems*.

[32] Paynter, H., 1961, *Analysis and Design of Engineering Systems*, MIT Press, Cambridge, MA.

[33] Turki, S., Soriano, T., 2005, "A SysML Extension for Bond Graphs Support," in *5th International Conference on Technology and Automation*, Thessaloniki, Greece.

[34] 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.

[35] 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.

[36] Czarnecki, K., Helsen, S., 2006, "Feature-Based Survey of Model Transformation Approaches," *IBM Systems Journal*, **45**(3), pp. 621-645.

[37] Object Management Group, 2007, "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification," http://www.omg.org/docs/ptc/07-07-07.pdf, April 2008.

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

[39] Königs, A., 2005, "Model Transformation with Triple Graph Grammars," in *Model Transformations in Practice, Satellite Workshop of MODELS 2005* Montego Bay, Jamaica.

[40] Keeney, R. L., 1994, "Creativity in Decision Making with Value-Focused Thinking," *Sloan Management Review*, 35(4), pp. 33-41.

[41] Fritzson, P., 2004, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, IEEE Press, Piscataway, NJ.

[42] University of Paderborn Software Engineering Group, 2007, *Fujaba Tool Suite 5*, http://wwwcs.uni-paderborn.de/cs/fujaba/, April 2008.

[43] Real-Time Systems Lab, 2007, *Moflon*, http://www.moflon.org/index.html, April 2008.

[44] Johnson, T. A., Paredis, C. J. J., and Kerzhner, A., 2008, "The SysML Transformers Plugin for Embedded Plus: A User's Guide," Georgia Institute of Technology, Atlanta, GA, http://srl.gatech.edu/Members/tjohnson/SysMLTransformers.zip, April 2008.

[45] EmbeddedPlus Engineering, 2007, *EmbeddedPlus SysML Toolkit*, http://www.embeddedplus.com/SysML.php, April 2008.

[46] Fritzson, P., et al., 2007, "OpenModelica System Documentation," http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/releases/1.4.3/doc/OpenModelicaSystem.pdf, April 2008.

[47] Dynasim, 2008, *Dymola 7.0*, http://www.dynasim.se/index.htm, April 2008.

[48] Nagel, L. W., and Pederson, D. O., 1973, "Spice (Simulation Program with Integrated Circuit Emphasis," University of California, Berkeley, CA

[49] Keeney, R. L., and Raiffa, H., 1976, *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*, Jon Wiley and Sons, New York.

[50] Paredis, C. J. J., 2008, *FluidPower Library for Modelica*.

[51] Pederson, K., Emblemsvåg, J., Bailey, R., Allen, J. K., and Mistree, F., 2000, "Validating Design Methods & Research: The Validation Square," in *ASME Design Engineering Technical Conferences*, AMSE, Baltimore, MD.

[52] Modelica Association, 2008, "Modelica Language Specification Version 3.0," http://www.modelica.org/documents/ModelicaSpec30.pdf, April 2008.

[53] Phoenix Integration, 2008, *ModelCenter v7.0*, http://www.phoenix-int.com/products/modelcenter.php, April 2008.

[54] Maplesoft, 2008, *Maple 11*, http://www.maplesoft.com/, April 2008.