A Model-Driven Engineering Framework for Computational Replicability and Reproducibility

by

Joseph Ledet

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 7, 2016

Keywords: Computational Reproducibility, Model Replicability, Model Transformation, Platform Independent Modeling

Copyright 2016 by Joseph Ledet

Approved by

Levent Yilmaz, Chair, Professor, Computer Science and Software Engineering
David Umphress, Associate Professor, Computer Science and Software Engineering
James Cross, Professor, Computer Science and Software Engineering
Jeffrey Overbey, Assistant Professor, Computer Science and Software Engineering
Halit Oğuztüzün, Professor, Computer Engineering, Middle East Technical University, Ankara, Turkey

Abstract

The scientific community currently suffers from a lack of verification of results obtained from many well-intentioned researchers. Part of the reason for this lack is the difficulty in reproducing experiments performed by these researchers. One of the leading causes of this problem is replicating the computer models used in these experiments. Also, some reviewers may not be familiar with the tools, modeling environments, and languages used by the original researchers. Currently, the solutions to this problem typically rely on the ability and willingness of the original researchers to completely capture all pertinent details of the experiment and the reviewers to have a great deal of familiarity with both the nature of the experiment and the tools used for experimentation including the source modeling environment. Unfortunately, if one or more of these aspects is lacking, then a reliable review of the original experiment may be extremely difficult if not impossible.

As such, a need has arisen for the ability to reliably transform computer models from one modeling environment to another and the maintenance of platform-neutral representations for use in future experiments. A process for transforming a model that is specific to a particular platform into a representation that is not dependent on a specific platform is necessary. Creating such a definition of the model can then be transformed into other environments for validation of the results obtained from simulations using the original model. We present a solution using a hybrid of two transformation technologies to facilitate the execution of such a process. We show how the process has been used to successfully produce a Platform Independent Model from the essential components of a model developed in a Platform specific environment. We present the details of how we confirmed that this Platform Independent representation can be used to generate a second model in the original platform and the results from executing this second model are the same as the original model;

thereby showing that the models represent the same model structurally. Additionally, we present the work done to develop a web-based user interface to perform the necessary model transformations. This process will be a useful tool to assist with the ability of researchers to develop and execute reproducible simulation experiments. By using this tool to confirm the results obtained from simulation experiments, the results of these experiments will be more reliable, generating increased credibility for the scientific community.

Acknowledgments

First, I would like to acknowledge Him through whom all things are possible, my Lord and my God: "Do not conform to the pattern of this world, but be transformed by the renewing of your mind." - Romans 12:2

I would like to express my deepest appreciation for all that my wife Ann has sacrificed as she encouraged me along this path. This degree was earned as much by her as it was by me, and possibly more so. I would also like to thank my children for all their attempts at delaying the completion of this work.

I dedicate this work to Joe Depa who was the first teacher who taught me how to think.

Additionally, I cannot even estimate the worth of all the mentoring and oversight provided by my co-advisors Drs. Yılmaz and Oğuztüzün; both of whom gave invaluable insight and direction. I have much appreciation for the time I was able to study under the tutelage of them in America and Turkey and much respect for them as men, engineers, and scientists.

To my committee, thank you for being an integral part of helping me navigate the doctoral program; Dr. Umphress, for initiating me into the program as the grad student "mom" and my first class lecturer; Dr. Cross, for being my second instructor and lecturing professor for my first GTA experience; and Dr. Overbey, for being approachable and the kind of teacher from whom students want to learn.

I would like to express my sincere thankfulness for the work done by my colleagues Sema Çam, B. Kaan Görür, and Orçun Dayıbaş. Without Sema and Kaan producing the "reverse" transformation, we never would have been able to validate the results of our forward transformation. I must thank Orçun for his work to create the user interface. Son olarak, Kaan ve Orçun'a Türkçe pratiğime yardımcı oldukları için çok teşekkür ederim. Çatı'da yediimiz öğle yemekleri benim için sonsuza dek çok değerli olacak.

Table of Contents

Abstract . . . ii

Acknowledgments . . . iv

List of Figures . . . viii

List of Tables . . . xii

List of Abbreviations . . . xiii

1 Introduction . . . 1

2 Background . . . 5

2.1 Reproducible Research . . . 5

2.2 Literature Review . . . 6

2.2.1 Model Replicability and Reproducibility . . . 6

2.2.2 Model Driven Engineering . . . 8

2.2.3 Model Transformation . . . 10

2.2.4 Tools . . . 14

2.2.5 OMG Standards for Model-Driven Engineering (MDE) . . . 16

3 Problem Statement and Proposed Solution . . . 19

3.1 Current Problem . . . 19

3.2 Proposed Solution . . . 21

4 A Framework and Process Model for Transformation-Driven Model Replication . . . 24

4.1 Case Study . . . 25

4.2 Platform Independent Environment . . . 27

4.3 Transformation Tools . . . 33

4.4 Overall Process . . . 34

5 Development of the Process Using the Case Study . . . 37

5.1 Meta-Models 38
5.1.1 Simulink Meta-model 39
5.1.2 Intermediate Meta-model 42
5.1.3 SysML Meta-model 45
5.2 Simulink Model 45
5.2.1 Simulink Structure 45
5.2.2 Simulink Settings 47
5.3 Intermediate Models 47
5.3.1 Default Property Values 49
5.3.2 Components with Properties 49
5.3.3 Adding Ports 50
5.3.4 Connecting Ports 51
5.3.5 Removing Superfluous Objects 51
5.3.6 Non-viewable SysML 52
5.4 ATL Rules 54
5.4.1 Default Property Values 54
5.4.2 Components with Properties 54
5.4.3 Adding Ports 55
5.4.4 Connecting Ports 56
5.4.5 Removing Superfluous Objects 57
5.4.6 Non-viewable SysML 58
5.5 XSLT Rules 59
5.5.1 Simulink Settings 59
5.5.2 Simulink Structure 60
6 Validation 61
6.1 Validation Methodology 61
6.1.1 Viewable SysML 61

6.1.2 Generating a SysML Model 62
6.1.3 Re-creating a Simulink Model 68
6.1.4 Validating Models 71
6.2 User Interface 74
6.2.1 Chaining Transformations 76
6.3 Graphical User Interface 77
6.4 Lessons Learned 81
6.4.1 Model Driven Engineering 81
6.4.2 Reverse Engineering 82
6.4.3 Creating a Hybrid Nature for Transformations 82
7 Conclusions 85
7.1 Benefits 85
7.1.1 Model Replicability 85
7.1.2 Model Driven Engineering 87
7.2 Future Work 88
7.2.1 Simulink MATLAB and SysML ACT Diagrams 88
7.2.2 State Machines in Simulink 90
7.2.3 Other Modeling Paradigms 91
Bibliography 93
Appendices 98
A ATL Examples 99
B XSLT Examples 114
C XML Examples 120
D ANT Task Example 126

List of Figures

2.1 Meta-meta-model, Meta-Model, Model, and Subject [7] 9

2.2 OMG definitions for layers of abstraction [39] 10

2.3 Model Transformation 11

2.4 Relationship between UML and SysML [8] 17

2.5 Diagram types available in SysML [8] 18

3.1 Transformation from/to one PSM to/from another PSM 20

3.2 Transformations from/to any PSM to/from any other PSM 21

3.3 Transformations from/to a PSM to/from a PIM 21

3.4 Transformations from a Simulink Model to SysML 23

4.1 MATLAB Simulink User Interface for Robot Soccer 27

4.2 MATLAB Simulink User Interface for Team 1 Strategy 28

4.3 Simulink GUI for Team 1 Strategy 30

4.4 SysML Block Definition Diagram for Team 1 Strategy 30

4.5 SysML Internal Block Diagram for Team 1 Strategy 31

4.6 Simulink Block Structure for Embedded Code Block 31

4.7 SysML Activity Flow Diagram for Team 1 Strategy 32

4.8 Simulink to SysML Transformation 35

5.1 Meta-Model of Model Section of SLX Formatted Simulink File 40

5.2 Meta-Model of Property Element of SLX Formatted Simulink File 41

5.3 Meta-Model of Model Section of SLX Formatted Simulink File 43

5.4 Simulink Structure for the Model Team 1 Strategy 49

5.5 Model Used as Source for Final ATL Transformation to SysML 50

5.6 Block Default Properties Structure 51

5.7 Main Structure of Team 1 Strategy 52

5.8 Intermediate Model after Adding Ports 53

5.9 Intermediate Model after Connecting Ports 54

6.1 Generated BDD for Team 1 Strategy 62

6.2 Generated IBD for Team 1 Strategy 62

6.3 Generated BDD for RoboSoccer 63

6.4 Generated IBD for RoboSoccer 63

6.5 Generated IBD for RoboSoccer 64

6.6 SysML BDD for RoboSoccer 64

6.7 SysML IBD for RoboSoccer 65

6.8 SysML IBD for Robot 1 Dynamics . . . 65
6.9 SysML IBD for Ball Dynamics . . . 66
6.10 SysML BDD for Team 1 Strategy . . . 66
6.11 SysML IBD for Team 1 Strategy . . . 67
6.12 SysML BDD for Team 2 Strategy . . . 67
6.13 SysML IBD for Team 2 Strategy . . . 68
6.14 Simulink GUI for Generated Team 1 Strategy . . . 69
6.15 Simulink GUI for Generated Team 1 Strategy Expanded . . . 69
6.16 Simulink GUI for Generated Team 1 Strategy MATLAB Code . . . 70
6.17 SysML for Team 1 Strategy with Modification . . . 73
6.18 Generated Simulink for Team 1 Strategy with Modification . . . 73
6.19 Modified Simulink for Team 1 Strategy . . . 74
6.20 Modified Simulink for Team 1 Strategy Transformed to SysML . . . 75
6.21 Modified Simulink for Team 1 Strategy to SysML to Simulink . . . 75
6.22 User Interface Main Page . . . 78
6.23 Selection of Transformation Type . . . 79
6.24 Select a Model to Transform . . . 79
6.25 Processing the Transformation . . . 80

6.26 Successful Transformation 80

7.1 Meta-Model of StateFlow Section of SLX Formatted Simulink File 89

7.2 Meta-Model of Activity in Intermediate Models 91

7.3 Meta-Model of Intermediate Models 92

List of Tables

2.1 Comparison of Model Transformation Tools and Technologies 15

4.1 Comparison of ATL and XSLT 34

5.1 Equivalent Objects in UML, SysML, and Simulink 45

6.1 Results of Simulations Using Original 72

6.2 Results of Simulations Using Generated Model 72

List of Abbreviations

AM3 Atlas MegaModel Management

AMMA ATLAS Model Management Architecture

ANT Another Neat Tool

ATL ATLAS Transformation Language

GME Generic Modeling Environment

GReAT Graph Rewriting and Transformation

GUI Graphical User Interface

MDA Model Driven Architecture

MDE Model Driven Engineering

MOF Meta Object Facility

OMG Object Management Group

PHP PHP: Hypertext Preprocessor

PIM Platform Independent Model

PSM Platform Specific Model

QVT Query/View/Transformation

SED-ML Simulation Experiment Markup Language

SysML Systems Modeling Language

SysML ACT	SysML Activity Diagram
SysML BDD	SysML Block Definition Diagram
SysML IBD	SysML Internal Block Diagram
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
XSLT	XSL Transformations

Chapter 1

Introduction

One of the hallmarks of scientific discovery is the ability to reproduce the results obtained from the execution of experiments [66]. This is true in a broad range of scientific disciplines. The growth of technology has enabled researchers from many different areas to perform experiments in a vast array of situations, many without having to be limited by physical resources. Scientists and researchers can replicate complex systems in a computer simulation environment and alter the inputs and properties to assess what affect these changes will have without having to implement each of these situations physically.

At the heart of scientific research is the ability for independent researchers to replicate the results obtained by the primary researchers. Referring to computations, this is known as "reproducible research" [42]. Both reproducibility, the reproduction of results in a subsequent run of a simulation, and replicability, the running of a simulation in a different modeling environment or formalism, can have a profound impact in the reliability of results from simulation experiments [67]. Unfortunately, many computational studies are not reproducible nor replicable [37]. One reason for this is the rarity with which researchers are able to share the details of the experiments. This is often due to researchers not recording all pertinent details in their simulation experiments [37]. Often, the details of the models developed and their respective simulations are not available to the broader scientific community for the purposes of verifying the correctness of the model and validating the results obtained.

Another factor in model reproduction is the simulation environment used to develop the desired model. There exist multiple options of modeling formalisms from which developers can choose. Some are structural and mathematically based, such as MATLAB Simulink;

others, RePast for example, are agent-based [25] [14]. While the differences between modeling environments can be useful to make the development of a model easier, they can also present challenges. Even when an original model is available, validation can still be a challenge if a second researcher does not have adequate understanding of the modeling environment used for simulation.

If a process to create the replication and/or reproduction of computer models used in experiments was more accessible, simulation experiments could be more easily reproduced. Thus, the results would be more defendable due to the ease with which they could be validated. On the other hand, some results could be shown to potentially be invalid. In both of these cases, the end result would be valid and trustworthy research [71]. The scientific community has attempted to somewhat validate itself by asking researchers to acknowledge when the results of others were not reproducible [49]. However, this is intended to be done after the initial results have been reviewed and published. Suggested solutions for achieving reproducible research have included using version control, reusing the modeling environment, and the development of automated capture tools [37]. However, many of these solutions rely on the independent researchers to have access to the original model and the modeling environment in which it was developed.

Model-Driven Engineering has emerged as a resource for enhancing development by representing components of the system developed, including the system itself, during the development process as models [32]. This is not solely limited to software development, but a broad range of engineering disciplines. However, MDE is not a standard, it is rather a set of guidelines or principles used to enhance system development processes. The onus of determining what specific tools or technologies to use falls on the development team. Fortunately, entities, such as the Object Management Group, have taken it upon themselves to develop a set of standards to facilitate development using MDE. By using the principles of MDE and making use of tools conforming to OMG standards, a usable solution to encourage reproducible research can be developed. Using model transformation and Platform

Independent Modeling, a representation of a model can be created that does not require the independent researcher to have knowledge of the original model's platform.

We implemented a process that transforms instances from Platform Specific Models represented in MATLAB Simulink into Platform Independent Models in SysML. We report on the successful implementation of an automated process that transforms well-formed models from a Simulink representation into an equivalent set of SysML diagrams. We detail the steps taken to develop this process, including the methodology used to validate the resulting models and lessons learned during the development process. Additionally, we describe the concurrent work to develop a process to transform SysML models into a PSM representation in Simulink. We present the current results from the development of this process using the case study model as well as the work done on a user interface for performing the transformations. Finally, we consider what remaining tasks are necessary for this process to be fully realized and beneficial to the scientific community as well as reasons that this process will be more useful than other solutions currently proposed for reducing the credibility gap in model-based scientific experimentation. Our solution will add wealth to the area of scientific research by providing a means to encourage reproducible research. By using Platform Independent Modeling along with a tool for automated experiment definitions and executions, we can overcome the challenges presented due to the differences among modeling platforms.

For example, we can consider a scenario in which the process or data of a simulation experiment had a major flaw, such as the case with Chang's team [60]. If the results of this experiment were verified by an independent researcher or team of researchers easily and quickly using an equivalent model to that used by the original researchers before final publication of the results, we could avoid many of the problems that occur when papers must be retracted and doubt is cast on the practices used to perform experiments. The reputations of the researchers on Chang's team could have remained intact as well as the confidence in the ability of science to identify and discover important truths about the nature of the universe.

We anticipate the work presented here to be used in a wide array of applications:

- Model exchange and longevity
- Tool interoperability
- Model replicability
  - Validation of experimental results
  - Experiment variability
  - Experiment management
  - Security of intellectual property
- Model Driven Engineering
  - Collaborative development
  - Software process

Chapter 2

Background

2.1 Reproducible Research

At the core of reliable scientific experimentation is the ability to perform experiments to better understand the nature of the universe. When these experiments are performed again and the results are the same or similar, we can have more confidence that the initial conclusions were and are correct. However, when results of subsequent experiments produce different results, our original conclusions must be modified or abandoned [56].

Unfortunately, often the focus of an experimenter can be limited to the development of a simulation model for experimentation [52]. This limits and can severely hinder the researcher's ability to focus on more desirable tasks such as analyzing data and results for future simulation runs. Therefore, often, a design process for the development of simulation models used in experiments is sorely lacking [63]. Much work exists to assist with the development of useful methodologies to create and execute simulations for experiments [51] [62]. These methodologies are very useful for the limiting of the number of simulation runs that might be unnecessary for the purposes of testing a given hypothesis. This is useful when a subsequent simulation for validation is executed in the same platform with the same setup as the first. However, often the reliability of results can be improved by performing the experiments in a second environment, especially if the second researcher has limited understanding of the original environment. The ability to transform models used for experiments into secondary platforms for execution would be extremely useful in closing the credibility gap in current scientific experimental results.

2.2 Literature Review

Prior to beginning any discussion of research involving model development, it is helpful to first have a concrete definition of a "model". Rothenberg defines the term thusly:

> Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality. [61]

2.2.1 Model Replicability and Reproducibility

One of the foundations of the scientific method is the ability to verify the results of experiments and extend the artifacts used in running the experiments. The ability to perform proper peer review is a necessity in producing accurate and valuable scientific research [43]. With the increased use of computer models and simulations in the performance of scientific experiments, the need for the trustworthy reproduction of the computational artifacts used for experimentation to generate credibility in the results obtained has become that much greater [71].

Model *reproducibility* takes an implementation of an existing computer model and reruns the simulation with the expectation of obtaining the same results [67]. Elements of the model, such as control variable values or run settings, can be altered to perform new experiments. Unfortunately, components of the original simulation experiment, such as the source code for models or the source data, are not always available. Even when these components are available, a second researcher intending to validate the results of the original experiment may not have intimate knowledge of the platform used for experimentation.

As a result, *reproducible research* as a leading factor in performing computational studies has arisen primarily due to a perceived lack of reliability in the results given in published research [42]. Unlike reproducibility, *replicability* is the development of a model or simulation that is distinct in some way from the original [37, 35]. While reproducibility can be accomplished by having access to the original computer model and data, replicability can be viewed more as an attempt to validate the original model's results by altering some aspect of the model used to perform the simulation. The advantage that replicability adds is that by incorporating an aspect of dissimilarity, creativity and alternative solution strategies are encouraged. In addition, this can be useful if a developer does not have access to or sufficient knowledge of the source model's environment. By having a second model in a different modeling paradigm, the ability to verify the results of the original is enhanced. If the developer of the second model has access to the original source model too soon in the process, the second model, if developed using the same modeling formalism or environment, may share too many characteristics with the original. Thus independent thought and methodology is lost or at least severely hindered. However, even with this difference, the original implementation and the second must be similar enough such that cross-validation occurs [72]. Without standards for maintaining documentation, validating, and distributing the details of computer models used in experiments and their respective results, reproducible research becomes much more difficult [66, 54]. The absence of such standards can be at least partially attributed to lax attitudes in performing reproducible research [40]. In addition, the lack of automated tools and methodologies encouraging reproducibility and replicability is a hindrance to performing reproducible research [37, 43]. As the use of computational artifacts in research grows, so will the need for developing and adhering to standards for sharing not only the results of experiments, but also the methods, tools, and source data for performing the experiments.

Existing suggested solutions for achieving reproducible research have included using version control, reusing the modeling environment, and the development of automated capture

tools [37]. However, many of these solutions rely on the independent researchers to have access to the original model and the modeling environment in which it was developed in order to adequately validate the results obtained. This reliance requires the original researcher or developer to not only capture all pertinent details of the simulation experiment, but also provide these details to the larger scientific community. This can often be a difficult undertaking [43]. This is often due to legal issues concerning intellectual property rights [66]. Executable papers, which involve the inclusion of executable code in the published papers, have also been presented as having potential to aide in the development of reproducible research [57]. Even when the source model is provided, the independent researcher must have insight into the nature of the original modeling environment.

Other solutions incorporate the use of workflows, such as Taverna and Kepler, which are sets of computational tasks organized in a coherent manner [12] [27]. Each task in the workflow is given inputs and generates outputs. These outputs are then given as inputs to the next task [68, 73]. However, workflow management can become problematic as the ability to re-execute the workflows and reproduce the initial results deteriorate over time [73].

2.2.2 Model Driven Engineering

Model Driven Engineering (MDE), a methodology for system development based on the use of modeling, meta-modeling, model transformation, platform independent modeling, and megamodeling, is increasingly becoming useful in developing and maintaining software systems [32]. By defining aspects of the system as models, the process of designing and implementing applications can be simplified [45].

Meta-Modeling

One of the hallmarks of MDE is the use of *meta-modeling* to define the structure of a domain model [64]. Each model can be considered to be an instance of the meta-model

defining the relationships among the components. Also, each level is connected strongly to the one above it in that any artifacts created at one level must conform to the definitions described in the level above it (i.e. a subject must conform to its model definition, a model must conform to its meta-model, etc.). However, the relationship between a model and its meta-model is not exactly the same as that between the subject and its model. Figure 2.1 shows that while the model represents the subject, a meta-model describes the configuration of a model [7]. While the number of modeling layers could be as many as a user defines, it is very rare that a system will be defined using more than four: Meta-meta-model, Meta-model, Model, and Subject or Object [58]. Figure 2.2 shows how some of the standards defined by OMG are used in these layers.

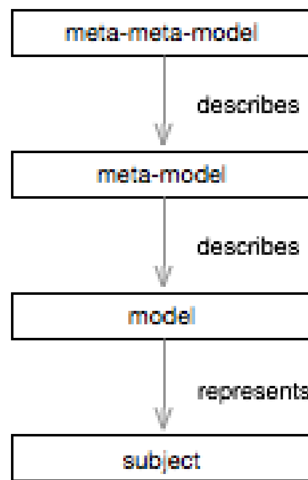

Figure 2.1: Meta-meta-model, Meta-Model, Model, and Subject [7]

Just as metadata can be defined as data about data, metamodeling can be defined as a model describing a model or set of models [30]. Metamodeling can be useful for defining multiple distinct models with shared characteristics [53]. Using the metamodeling concept, researchers can better understand and evaluate data models [41]. It can also be useful in implementing a model transformation process [34]. Also useful is the meta-metamodel construct. Just as a meta-model defines the objects and relationships of a model, a meta-metamodel defines a meta-model [30].

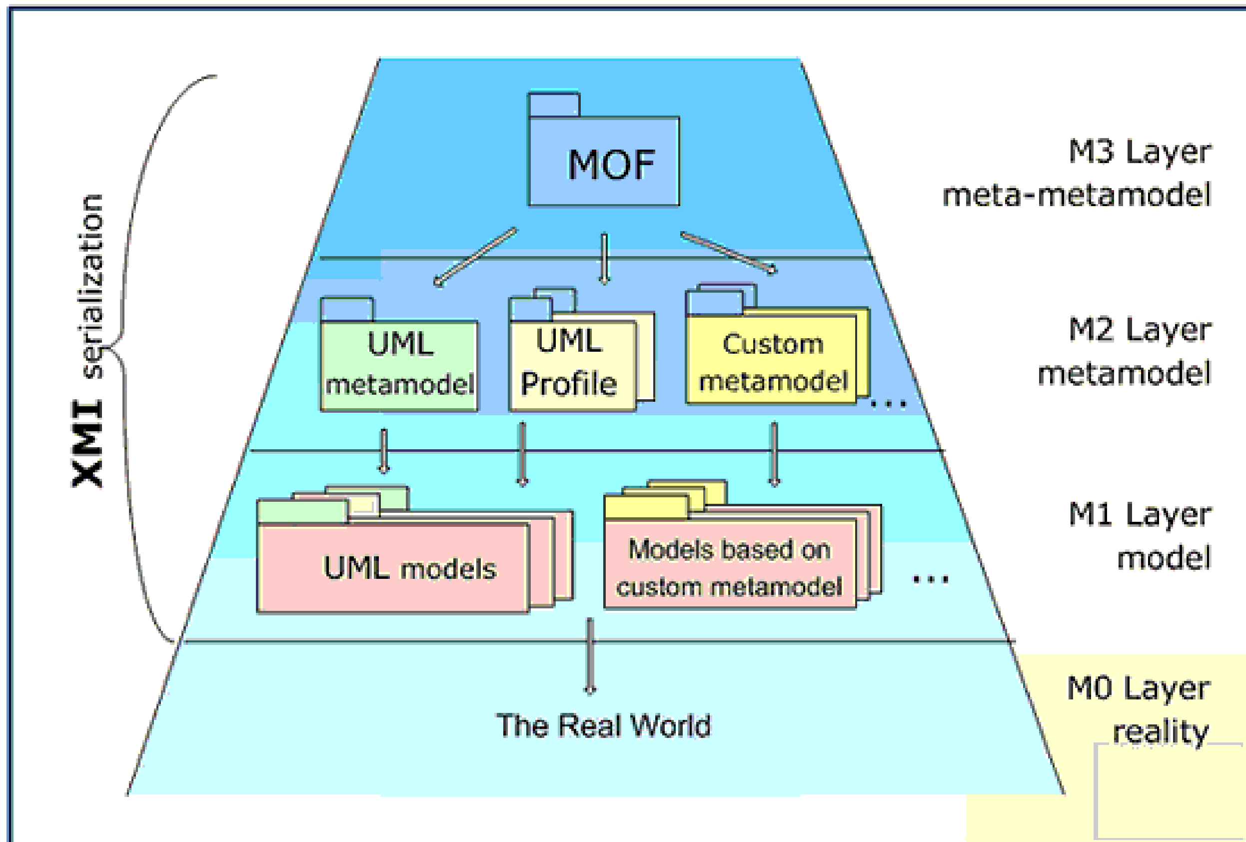

Figure 2.2: OMG definitions for layers of abstraction [39]

A strong understanding of meta-modeling is a key to performing a successful model transformation. By defining the components of a model at a higher level of abstraction, a developer can more easily define the best way to convert the details of the model into a one existing in a second environment.

2.2.3 Model Transformation

The layers of abstraction defined in Section 2.2.2 can be very useful for performing a *model transformation* from a source model to a target model. A transformation can be done between two different levels of abstraction. For example, a transformation can be performed on a meta-model describing the abstract syntax of a language, creating a model written in code of the language desired (i.e. code generation) [38]. When making this type of transformation, additional information must be supplied to the transformation process.

Another form of transformation can be between two different models at the same level abstraction in two different modeling languages. In both of these types of transformations, a set of rules must be defined for how to get from the source model to the target model. Figure 2.3 shows the relationships between the various models used in a transformation. If the meta-model and the model definitions for a source model are given along with the meta-model definition for a target model, then a set of rules can be defined to transform the source model into a target model that conforms to the targets meta-model. A transformation can also be done between a Platform Independent Model (PIM) and a Platform Specific Model (PSM).

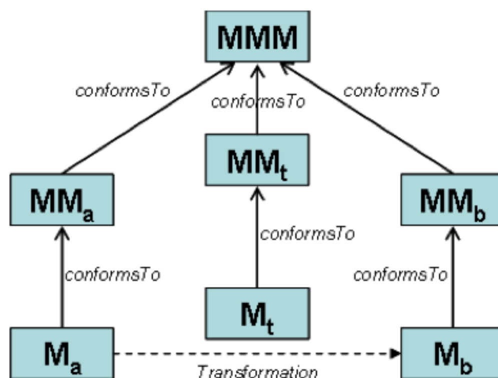

Figure 2.3: Model Transformation

TGGs

Triple Graph Grammars (TGGs), as described by Kindler and Wagner [50], are useful for defining a relationship between two distinct types of models. These can thus be used to transform a model from one type to another. Some potential problems with using TGGs to perform transformations are given:

1. Some source aspects cannot be transformed into a target formalism. Therefore, a complete transformation into another model would not be possible.

2. A source object can possibly have multiple model representations. Therefore, a transformation in this situation would not be deterministic.

3. When transforming from the source formalism to the target representation, some rules may offer multiple choices. Choosing one option may result in a dead-end, whereas other options would result in a successful transformation. Therefore, there may be a need at times to revert to a previous state during the transformation process. As such, the process would be inefficient.

Query/View/Transformation (QVT)

The Query/Views/Transformations Request for Proposals (QVT RFP) was established by the Object Management Group (OMG) as a means to standardize the methods used to define model transformations involving modeling languages that are part of the Meta Object Facility (MOF) [65].

Certain characteristics are associated with such transformations:

1. Multiple domains, each identifying a candidate model. Relations are satisfied by locating, modifying, or creating all properties and associations

2. Relation domain defines the type of relation between different domains. Checkonly, as its name implies, checks for the existence of a valid match for the relation to be satisfied. Enforced means the relation is satisfied by creating, modifying or deleting the necessary elements.

3. The when clause defines conditions that must be exist prior to executing the transformation.

4. The where clause defines conditions which must be satisfied by all model elements once the transformation is executed.

5. Relation types

    (a) Top-level: all of these will be fulfilled once the transformation is complete

    (b) Non-top-level: will be fulfilled when invoked from another relation's where clause

QVT consists of three languages [48]:

1. *Relations*
   (a) Declarative language is used
   (b) Concrete, abstract, and graphical syntax is available
   (c) Set of relations specifies transformations
   (d) Object patterns make up relations
   (e) Object patterns are used to instantiate necessary new model elements or make necessary changes to existing model elements
   (f) Details of the transformation are hidden from the developer
   (g) The *RelationsToCore* transformation can be created with relations
2. *Core*
   (a) Declarative language is used
   (b) Concrete and abstract syntax is available
   (c) Transformation definitions are generally longer than those in *Relations*
   (d) Developer must create and maintain transformation details
   (e) Core provides basis for specification of the *Relations* language semantics
3. *Operational Mappings*
   (a) Imperative language is used
   (b) Mapping extends *Relations* and *Core* languages
   (c) Relations are specified declaratively can be imperatively implemented

2.2.4 Tools

The Eclipse Modeling Framework (EMF) project is licensed under the Eclipse Public License. In addition to being a modeling framework, it is a code generation utility used to build tools and applications using a data model as its basis [6]. ATLAS Model Management Architecture (AMMA) is also a framework for developing models [33]. AMMA is part of the Eclipse project and is built onto EMF. AMMA uses ATL to facilitate transformations. The Generic Modeling Environment (GME) is used to create domain-specific models using meta-models [36]. Graph Rewriting and Transformation (GReAT) is a tool for creating other tools to transform models using methods also used to transform graphs [31]. Table 2.1 compares some of the primary features of these tools.

The ATLAS Transformation Language (ATL) began with the same initial specifications as QVT [48]. As such, there are many similarities between these two model transformation languages. Some of the primary characteristics of ATL are as follows:

1. Declarative part consists of rules which creates target model elements for every matching source model element

2. Two imperative parts

   (a) Explicitly called rules which may consist of a declarative target pattern

   (b) Blocks of imperative instructions

3. Ideally, only declarative rules should be used

4. Transformations are done in one direction only

Platform Independent Modeling

Part of the modeling aspect set forth in MDE, is the concept of *Platform Independent Modeling* (PIM) [55]. A Platform Independent Model, as its name suggests, is a model

Tool or Technology	Licensed/ Developed by	Operating Systems Supported	Transformation Tool
EMF	Eclipse	Windows Linux Mac OSX	No
GME	ISIS (Vanderbilt)	Windows	No
XML/XMI	W3C	Windows Linux Mac OSX	No
AMMA/ATL	Eclipse	Windows Linux Mac OSX	Yes
GReAT	ISIS (Vanderbilt)	Windows	Yes
XSLT	W3C	Windows Linux Mac OSX	Yes

Table 2.1: Comparison of Model Transformation Tools and Technologies

that does not rely on a specific platform. Nor does it contain information related to the technology used to develop it [65]. A Platform Specific Model (PSM), on the other hand, is a model that is developed using a specific platform or modeling environment. A PSM can be created from a PIM by transforming the PIM using platform and development technology specific information. Creating Platform Independent Models that are of high quality can, and should, lead to a creation of inevitably high-quality Platform Specific Models [55]. For example, a PIM could be developed using a graphical visualization tool. It would represent the high-level ideas of a given model and may indicate the anticipated behavior of the model when it is run. Since this model is not dependent on a particular platform or environment, there are very few, if any, constraints on syntax or the structure this model takes. This model would then be transformed into a PSM using the platform the developer chooses, such as Simulink [25] or RePast [14]. In a sense, this transformation reduces the level of abstraction of the given model, but it would be erroneous to view the PIM as a meta-model of the PSM.

Megamodeling

A *megamodel* is a tool or system for maintaining various aspects related to the development of the models used in MDE. Through the use of metadata pertaining to the models a megamodel references the pertinent models and meta-models in a centralized format [33]. The megamodel can also contain information dealing with the modeling environments and runtime configurations for the models in the system.

2.2.5 OMG Standards for Model-Driven Engineering (MDE)

The Object Management Group (OMG) maintains a set of standards for use in implementing MDE [22]. These standards MDA are defined in this section.

Model Driven Architecture

Model Driven Architecture (MDA) is OMG's standard for implementing the methodologies that are part of MDE [20]. MDA is the umbrella under which the other OMG standards for implementing MDE are maintained.

Meta Object Facility

The Meta Object Facility (MOF) supplies a standard for defining aspects of meta-models [58]. As such, any model developed using MDA must conform to a meta-model that itself conforms to the MOF standard.

Unified Modeling Language

The Unified Modeling Language (UML) is the OMG standard for providing a visual representation for models, including meta-models [59]. UML uses Classes, Generalizations, and Associations to define aspects of a system and the relationships between those components.

Systems Modeling Language

The Systems Modeling Language (SysML), which was developed by OMG, is a general-purpose adaptation of UML [15] [47] [46]. SysML can be used and distributed through an open source license. It makes use of visual tools to promote various activities related to systems engineering. While UML tends to be related mostly to software systems, SysML can also be used for other types of systems including hardware.

While it is not accurate to describe SysML as a subset of UML, the two standards are strongly related. SysML contains many of the same element types as UML; SysML Blocks are based on UML Classes [15]. SysML also includes aspects that are not present in UML, such as Requirements Diagrams. The relationship between UML and SysML is shown in Figure 2.4. The diagram types available in SysML are can be found in Figure 2.5.

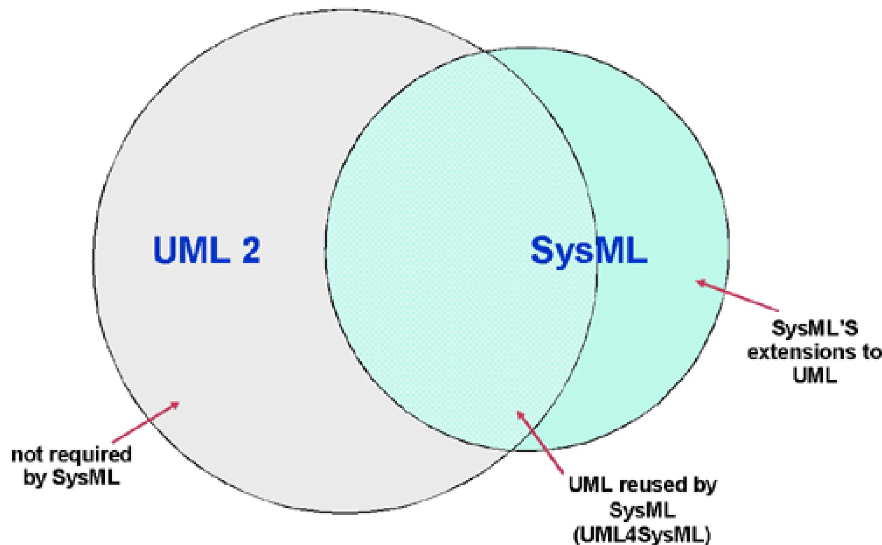

Figure 2.4: Relationship between UML and SysML [8]

XML Metadata Interchange

XML Metadata Interchange (XMI) uses XML syntax to store and exchange metadata for models. For example, models developed in the Eclipse Modeling Framework (EMF) are stored using XMI. The Extensible Markup Language (XML) was developed by the World

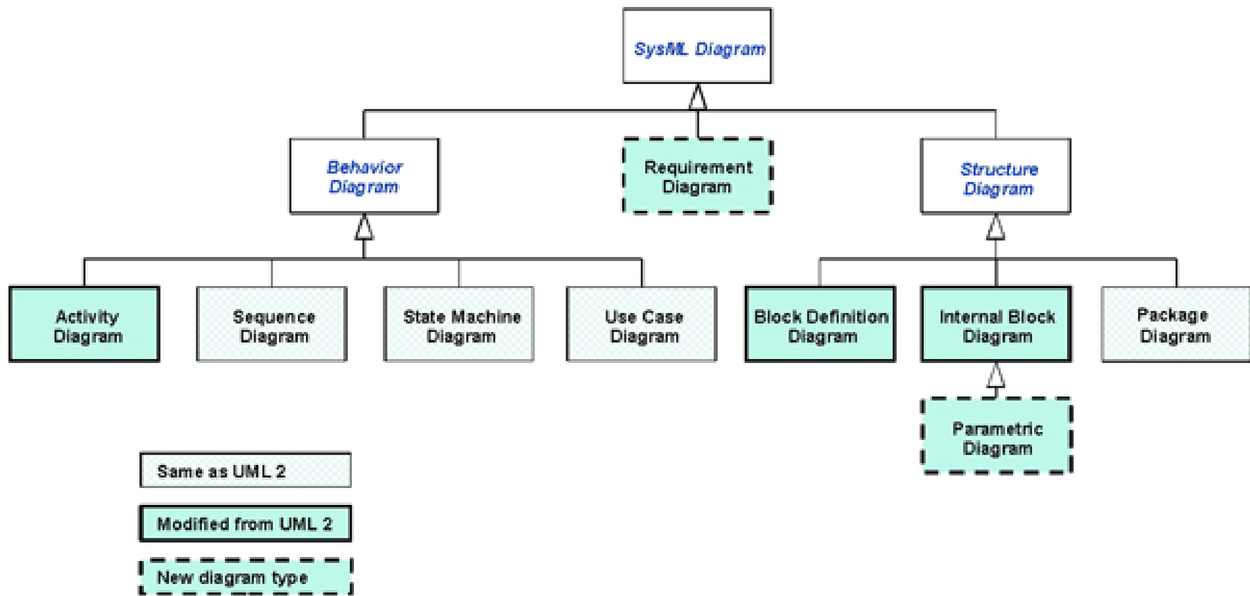

Figure 2.5: Diagram types available in SysML [8]

Wide Web Consortium, but is used in many document formats including the basis for XMI [11].

Query/View/Transformation

The details of the Query/View/Transformation (QVT) standard can be found in Section 2.2.3.

Megamodel

OMG does not currently support a standard for megamodeling. However, options do exist for researchers wishing to implement this system as part of an MDE development process:

- AMMA includes the Atlas MegaModel Management (AM3) tool to fulfill the needs for a megamodel [33]
- Waltemath et al. have implemented the Simulation Experiment Description Markup Language (SED-ML) as a solution to the need for a megamodeling tool [69]

Chapter 3

Problem Statement and Proposed Solution

3.1 Current Problem

Currently, there exists a lack of substantial development of an MDE process for the automated creation of PIMs from existing PSMs for the purposes of enhancing model replicability through model transformation. More effort seems to have been given to the development of processes to transform a given PIM into a PSM [55] [34] or directly from a specific source platform to a specific target platform [70].

The usefulness of transforming from a PIM to a PSM lies in the ability for a developer to design a model without specific knowledge of any particular language. This allows the developer to concentrate more on aspects of the model design without dealing with issues related to platform specific notions such as syntax or compilation errors. However, this process relies on the PIM or its detailed design being readily available. This is not always the case, as often models exist solely as source code in a particular language. The usefulness of transforming directly from one platform to another lies in the ability to create rules for transformation that only take the particular structures of the two languages (source and target) into account. There is not a need to account for any possible language into which a future developer may want to transform the model. However, this is also one of the disadvantages of this approach. For the purposes of the reliability of experiments using models, this would mean that researchers wishing to reproduce the results of the experiment would need to have access to either the platform of the original model or a well-formed set of transformation rules into a platform to which the researcher has access. If this set of rules does not exist, the researchers would be required to define these rules or re-create the model in the target platform themselves. This could be very time-consuming and may

distract researchers from other important tasks, such as defining variables and constants or analyzing results of the experiments. Another disadvantage is related to the method used to perform these transformations. Often these transformations are executed on a particular model; they are not general rules that could be applied to any model in the source language. Also, these rules are not necessarily bi-directional; even if a model is able to be transformed from one environment to another, a transformation executed from the second environment to the first might not yet be possible. Finally, even if a set of generic rules were developed from one platform into another, this would mean that in order to facilitate bi-directionality, two sets of rules would need to be developed. This can be seen graphically in Figure 3.1. However, to expand the ability to collaborate to any additional platforms would require the development of two sets of transformation rules between each of the platforms. For n platforms, n^2 - n sets of transformations will be necessary as seen in Figure 3.2. However, if a set of transformations to/from a PIM is implemented, as shown in Figure 3.3, then only $2n$ sets of transformations will be necessary. Additionally, if a modification or update is made to the nature of a platform, only the two rule sets for transformation to/from the PIM must be modified to accommodate this change. In a system that implements bi-directional rule sets between each platform, modification to $2n$ rule sets will be needed. As more platforms are used in a wide array of applications, it will become much more useful to develop and maintain the smaller number of transformation rule sets necessary.

Figure 3.1: Transformation from/to one PSM to/from another PSM

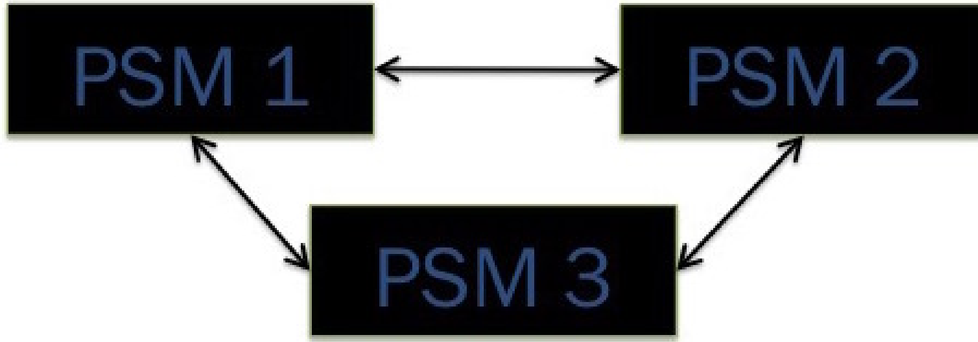

Figure 3.2: Transformations from/to any PSM to/from any other PSM

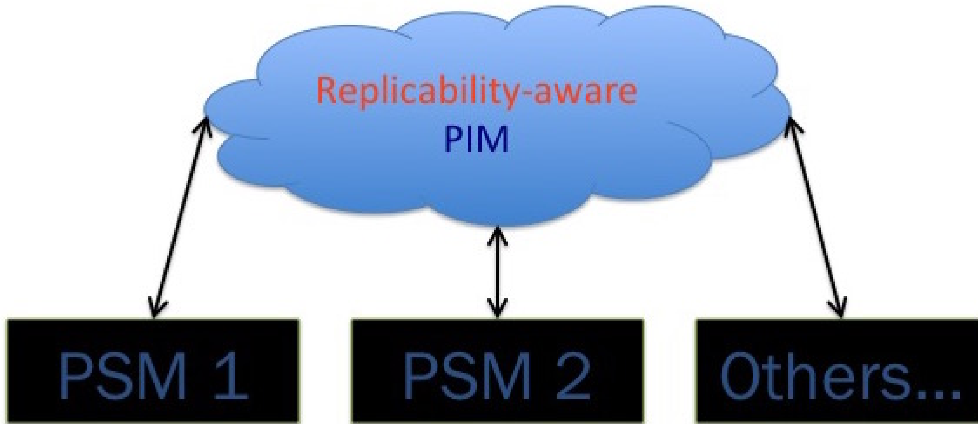

Figure 3.3: Transformations from/to a PSM to/from a PIM

3.2 Proposed Solution

We proposed and developed a set of rules to transform a model in a particular language into an XML-based platform-neutral representation. This representation can be used to generate a model in a second platform that conforms to the design of the original model. By transforming to an intermediate PIM, we will be able to have access to the usefulness of having a PIM even when the only representation currently available is a PSM. We utilized a reverse engineering process to make the transformation rules as generic as possible so as to create a set of conventions that could be used on any model in the source language. Additionally, we utilized lessons learned in the PSM to PIM process to create a second set of rules to transform from the PIM representation to an executable version in the original

platform, thus making our transformations bi-directional while concurrently validating our forward transformation process.

Another critical component of our research was to incorporate this model transformation process as part of a user interface for use by scientists and researchers. By being included in such a system, the PIMs will be reused for future experiments whether using the same modeling platform as the original or a second paradigm. Thus, this system will greatly improve the ability to enhance reproducibility and replicability for the models used in our process.

We analyzed a case study model in the source environment with the goal to develop a PIM representing this model. Once the PIM was developed, we were able to assist in the development of two other processes; one to transform the model back into the source domain and another to create an equivalent model in a second domain. The source domain and the second domain chosen are structurally different, such as a structural based modeling paradigm [25] and an agent-based modeling environment [14]. Once these two transformations are completely finished, we hope to have the backbone of two sets of rules; one for transforming to/from the source model environment and another for transforming to/from the target domain.

The task of developing our process required a great deal of time and resources to complete. Challenges that had to be overcome were not few, nor were they trivial. First, we selected the environments for source and target models, the format for PIMs, and the tools for performing transformations. Once these selections were made, we analyzed models in the source domain to understand the nature of the source modeling environment due to the limited information available on the general structure of the format of these models, such as the primary components of a model developed in this platform and interactions among these components. We also analyzed the nature of the PIM format. This required researching tools for displaying this format's information and understanding the tool's file formatting.

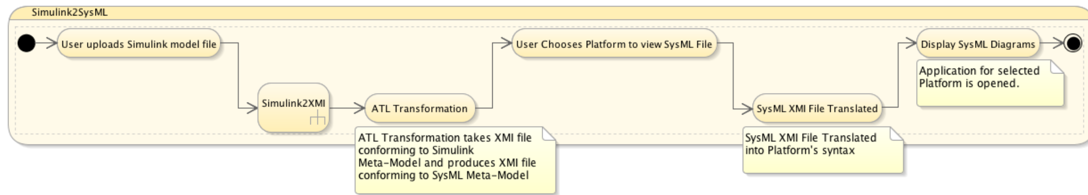

Figure 3.4: Transformations from a Simulink Model to SysML

Understanding the syntax for the tool for developing the transformation rules was absolutely essential. Also, understanding the format for the source model of a transformation using this tool was necessary so that we were able to develop a process for translating the source model into this format. Actually, this understanding resulted in the need to develop a transformation using a second transformation technology leading to a hybrid approach for model transformation. Finally, for transforming into PIMs, we determined the required components that are needed for performing transformations, such as meta-models for the source and target environments.

Once the first phase of this process was complete, the processes to perform transformations into a second environment and reverse transformations to the source environment did not require as many additional components. Since the tools for developing the rules were the same for all transformations, the necessary understanding for the target environment was limited to understanding the structure of the components of a model developed in the environment, the meta-model for the target domain, and a set of rules for transformation. The only requirements for performing reverse transformations was an additional set of transformation rules and an understanding of which components or attributes in the original model were deemed to be essential for execution.

Now that we have a process for performing pertinent transformations, a storage location for model files was also required, a server location and structure for storing all models, specifically PIMs, meta-models, as well as any scripts that must be run as part of the transformation process.

The potential user experience for performing transformations is presented in Figure 3.4.

Chapter 4

A Framework and Process Model for Transformation-Driven Model Replication

We used the MDE principles described in Section 2.2.2 as the foundation of a process to transform PSMs developed using MATLAB Simulink into PIM representations [25]. In this chapter, we detail the decisions made to facilitate the development of the transformation process. The generalities for the necessary decisions to be made are given below:

1. A case study for developing a transformation process that could be expanded. For the case study to serve as a valuable learning experience for the initial development of the transformation process, it needed to have the following characteristics:
   - The case study needed to include a wide range of features available in the Source environment
   - Also have simulation results such that it is trivial to validate that the replicate is sufficiently similar
2. A standard or technology for representing the PIM representation. The requirements for this standard were set as follows:
   - It was necessary that the standard be widely accepted by the engineering and scientific community
   - It should create models that are able to be accessed, viewed and edited without requiring a particular executable modeling platform or technology
3. A set of transformation tools and/or languages for transforming PSMs in the Source environment into PIMs in the technology chosen. These transformation tools were chosen to fit the following:

- They did not have steep learning curves
- It is relatively simple to perform a transformation; a script from the command prompt or a button click can execute the transformations

4.1 Case Study

In order to gain a better understanding of how Simulink models can be transformed into equivalent PIMs, we use a case study that assisted in our ability to become familiar with some of the key features of Simulink. By using an example as a case study that utilizes a broad range of Simulink features, we are able to gather enough information about Simulink models to establish a solid foundation for a process for transforming Simulink models. We selected the Robot Soccer [1] model as our case study. This model is an example model available via Mathworks that is designed to utilize key features in Simulink:

- Incorporates 19 distinct Simulink block types
- Includes Simulink System Blocks
- Utilizes model references to other Simulink model files
- Submodels have input and output ports
- Has S-Function blocks with embedded MATLAB code
- Does not use random number generators so it would be a simpler process for verifying correctness

The package for this model consists of three Simulink model files:

1. Robot Soccer - Figure 4.1
    - Main model file

[1] http://www.mathworks.com/matlabcentral/fileexchange/28196

- Simulates two teams of two robots each and a ball
- User can edit value for robot stamina. This will affect the robot's battery power and, thus, it's agility.
- Simulation runs until one team scores a goal

2. Team 1 Strategy
    - Submodel
    - Calculates actuator force for robots for team 1 based on robot's position relative to ball and ball's position relative to goal
    - Developer can alter values for multiplication factors to increase likelihood of team 1 scoring a goal
3. Team 2 Strategy
    - Similar to Team 1 Strategy, with different multiplication factors

Now that we have successfully completed the automated process to produce a PIM equivalent to the case study model, we have a solid foundation for completing the expanding of our process to cover features of Simulink that are not used in the case study model:

- Support for all 284 Simulink block types - the 19 block types incorporated in the case study represent approximately 7% of all available in Simulink [28]
    - 15 of the block types in the case study are in the list of the 23 "Commonly Used Blocks", meaning that the case study incorporates 65% of these block types [1]
- Support for state transitions - the stateflow portions of the case study model files utilize embedded MATLAB code, but do not include state machine transitions

[1] 26 block types appear in the list of "Commonly Used Blocks". However 4 of these (Add, Subtract, Sum of Elements, and Sum) are all represented in the Simulink model file as the type Sum. Since Robot Soccer uses a Sum block, we are confident that these other 3 block types should reasonably be considered as analyzed for our transformation process.

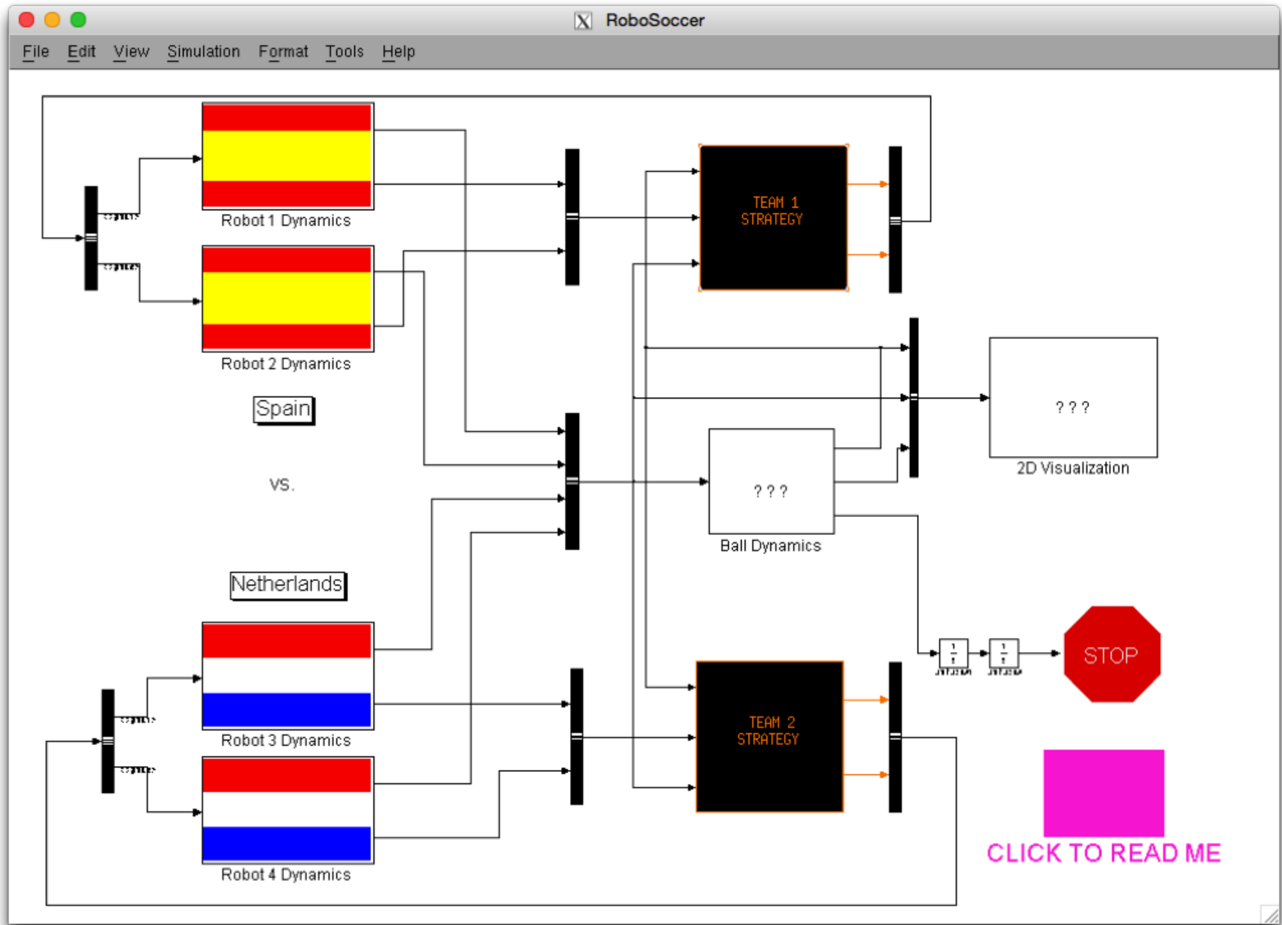

Figure 4.1: MATLAB Simulink User Interface for Robot Soccer

4.2 Platform Independent Environment

A sufficient platform independent environment will have components that are able to represent the elements of the source model. As stated above, the modeling paradigm selected should be a widely accepted standard in the field of science and engineering. Because of the reasons presented below, we feel that SysML is the strongest candidate to represent our PIMs:

- Like Simulink, SysML uses blocks, ports, and connections to represent the flow of data
- Like UML, SysML is considered an industry standard for model development

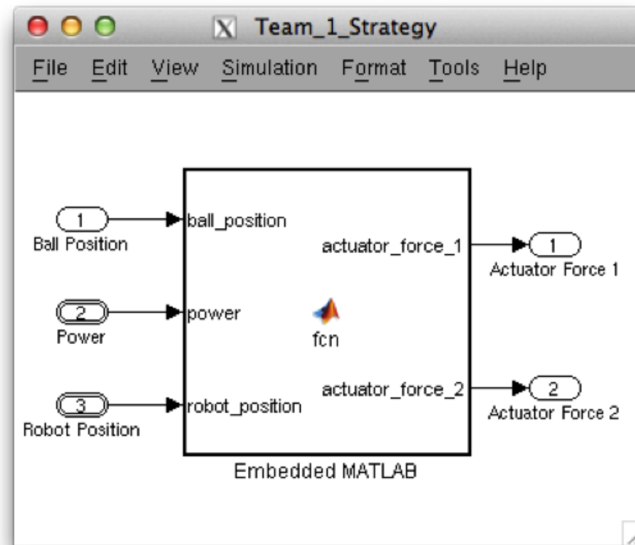

Figure 4.2: MATLAB Simulink User Interface for Team 1 Strategy

- SysML uses types of diagrams not available in UML, such as the Requirements Diagram, that are useful in the development life cycle
- Multiple tools exist to view and create SysML diagrams
  - Eclipse - Papyrus [18]
  - Rational Rhapsody [23]
  - Modelio [21]
  - Visual Paradigm [26]

As an example, we have used the Eclipse tool Papyrus to generate the expected SysML representation of the model information contained in the Team 1 Strategy Simulink file discussed in Section 4.1. This model includes three inputs (Simulink Inport Blocks), two outputs (Simulink Outport Blocks), and an Embedded Code Block (Simulink S-Function Block) with MATLAB code embedded. The Simulink GUI representation of this example is given in Figure 4.3. This model will reasonably be represented by the SysML Block Definition Diagram and Internal Block Diagram shown in Figures 4.4 and 4.5, respectively.

The Simulink blocks type of Inport and Outport represent inputs to and outputs from a System, respectively. The System can be the root System for the model or a Subsystem within a Subsystem block. These blocks will easily translate as Flowports of type *in* and *out* belonging to a Block in a SysML diagram. As can be seen, the IBD diagram in the SysML representation looks very similar to the Simulink GUI source. However, in the source model file for the Simulink S-Function block we find additional Simulink blocks. The general structure of the source model is as follows:

- GUI S-Function Block is represented in the XML model as a Simulink Block of type "SubSystem"
- This block has a Simulink System with the following structure:
  - One Simulink Block of type "S-Function"
    - Has a Property called "Tag" that connects to the Stateflow section for the MATLAB code behind the block
    - Has n input ports where n is the number of inputs to the GUI S-Function Block
    - Has $m+1$ output ports where m is the number of outputs from the GUI S-Function Block
  - n Simulink Blocks of type "Inport"
    - Each has Port Property with value i
    - Each has its sole outport connected to the inport of the Simulink S-Function Block with number i
  - m Simulink Blocks of type "Outport"
    - Each has Port Property with value k
    - Each has its sole inport connected to the outport of the Simulink S-Function Block with number $k+1$

– One Simulink Block of type ”Demux” with its sole inport connected to outport number 1 of the Simulink S-Function Block

– One Simulink Block of type ”Terminator” with its sole inport connected to sole outport of the Demux Block

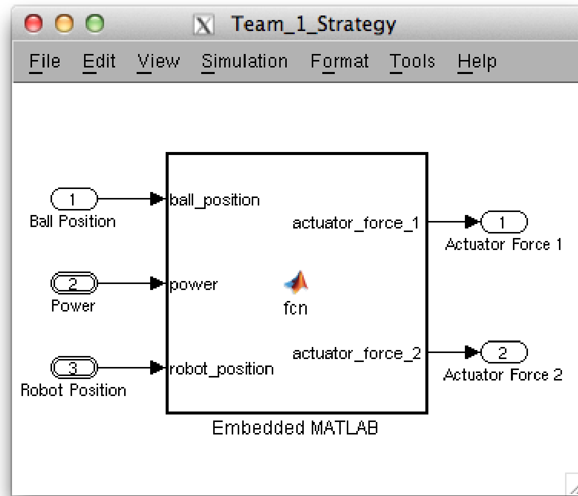

Figure 4.3: Simulink GUI for Team 1 Strategy

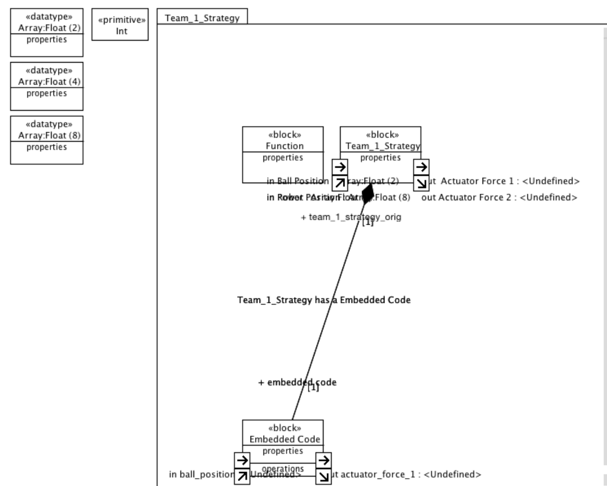

Figure 4.4: SysML Block Definition Diagram for Team 1 Strategy

This format is visually represented using the Team 1 Strategy example in Figure 4.6. As such, the transformation rules defined by the tool selected necessarily have a check for

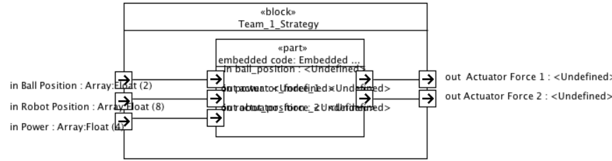

Figure 4.5: SysML Internal Block Diagram for Team 1 Strategy

this given structure and transform the block properly. When this situation is encountered, the transformation results in a single SysML block with n Ports of type *in* and m Ports of type *out*.

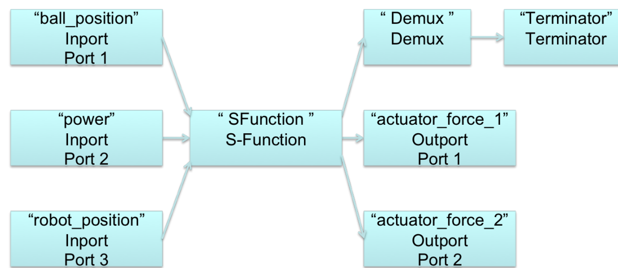

Figure 4.6: Simulink Block Structure for Embedded Code Block

The source MATLAB code from the Embedded Code Block is given in Listing 4.1. This algorithm would be transformed as a SysML Activity Flow diagram as shown in Figure 4.7. As of this writing, the strategy for transforming MATLAB code into a SysML Activity Flow Diagram is being discussed and a summary of current plans will be presented in Chapter 7.

Listing 4.1: MATLAB Source for Team 1 Strategy Model

```
function [actuator_force_1, actuator_force_2] =
fcn(ball_position, power,  robot_position)
%#eml


d1=(robot_position(1:2,1)-ball_position);
d2=(robot_position(3:4,1)-ball_position);
if dot(d1,d1)>5
   actuator_force_1=-22*d1;
else
   r1=(ball_position- [200;50]);
   actuator_force_1=-11*r1;
end
if dot(d2,d2)>5
   actuator_force_2=-25*d2;
else
   r2=(ball_position-[200;50]);
   actuator_force_2=-15*r2;
end
```

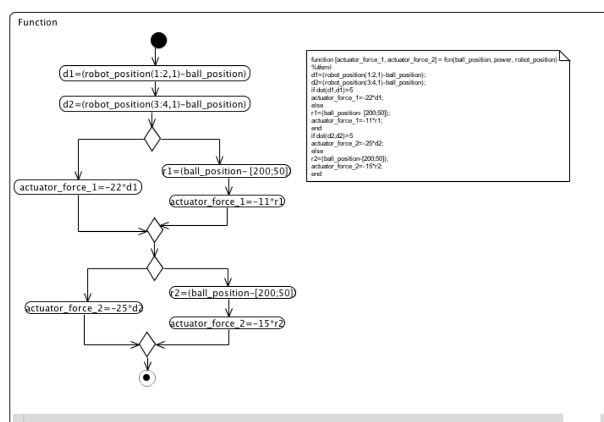

Figure 4.7: SysML Activity Flow Diagram for Team 1 Strategy

The example SysML BDD, IBD, and ACT diagrams shown in Figures 4.4, 4.5, and 4.7, respectively are used as a means of verifying that the process accurately produces valid and expected SysML representations of our case study Simulink model.

4.3 Transformation Tools

To facilitate the development of transformation rules and their subsequent application on a source model, we make use of the Atlas Transformation Language [17]. This tool presents the following benefits:

- Supported by Eclipse Modeling Framework with access to tutorials [16]
- Simple process for performing transformation once rules are defined
    1. Provide meta-models for source and target and xmi-based model representation
    2. Set run configuration with the locations of these elements
    3. Run transformation in Eclipse
- All three meta-models (source, target, and ATL rules) conform to ECORE meta-meta-model

ECORE is the format for defining meta-models in EMF [19]. By using ATL for transformations, we are required to define the meta-models for the source and target of any ATL-based transformation. The transformation also requires an XMI formatted source model that conforms to the source ECORE meta-model. Our original plans were to create a simple parser that would generate the XMI model. However, once it was discovered that the format for Simulink models saved with the SLX [1] included an XML-formatted file with the definitions of the model elements, we now utilize a second model transformation technology in the form of Extensible Stylesheet Language Transformations. A comparison of these two specific transformation technologies is shown in Table 4.1.

[1] From Simulink version R2013a and on, this has been the default format [24].

ATL	XSLT
Defined metamodels	No requirement for metamodels
Source model conforms to metamodel	Any source model is valid
Exceptions halt target model generation	Incorrect transformation rules produce unanticipated results
Each source object with a rule produces its target object(s)	Source objects must be explicitly transformed
Recursion difficult	Recursion easy
Looping over collections possible	Looping not possible

Table 4.1: Comparison of ATL and XSLT

As noted, XSLT does not require explicitly defined metamodels. While this gives the benefit of not needing separate metamodel definitions for transformations using XSLT, it also gives the potential for unanticipated results when the final model is produced. For example, the output of an XSLT transformation may produce an element that does not conform to the metamodel of an input to an ATL transformation. When the ATL transformation is attempted, an error will be encountered. Thus, more attention to detail must be used when writing the transformation rules when utilizing XSLT.

4.4 Overall Process

Using the decisions of SysML and an ATL/XSLT hybrid for the PIM environment and transformation utilities, respectively, the diagram shown in Figure 2.3 can be modified to specify these choices. The resulting transformation process is shown in Figure 4.8. The items highlighted in orange represent components that must be developed by hand. The item highlighted in darker blue is the XML file obtained from a Simulink SLX model file. This model will be used as an input to an XSLT transformation to obtain an XMI version of the block diagram to be used in the subsequent ATL transformation. Additionally, the nature of the Simulink source code required us to parse the text of the properties of certain Simulink elements to determine how blocks in a Simulink model are connected to each other and to embedded MATLAB code.

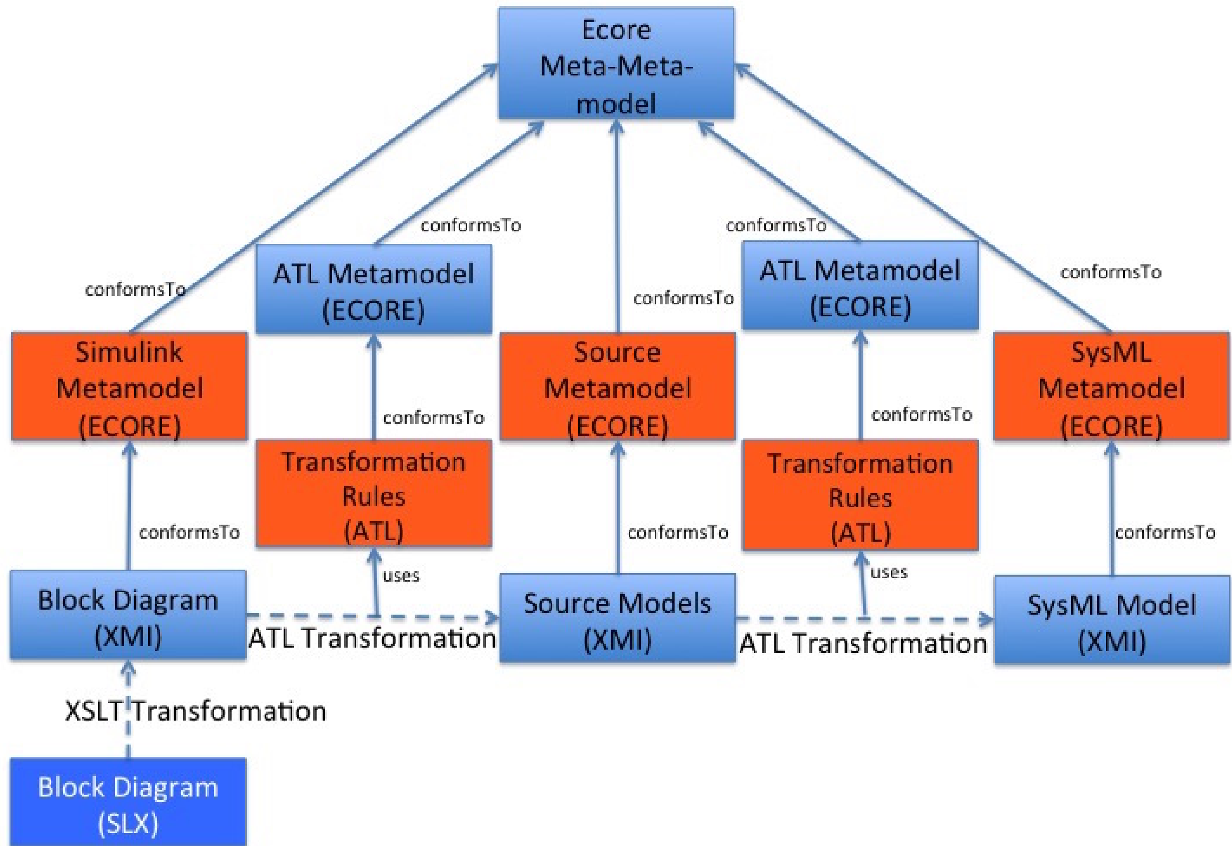

Figure 4.8: Simulink to SysML Transformation

Note that the middle transformed node is plural. This is because we decided to create a series of transformation processes, each with their own purposes. Each of these resulting intermediate models conform to the ”Source” metamodel. This decision had the following benefits:

- Allowed us to better learn the process of creating ATL and XSLT rules
- Provided a better means for debugging the process if unexpected results are observed in the PIM representation
- Was necessary for the accurate transformation of certain Simulink objects into SysML counterparts

Once all of the individual XSLT and ATL transformations had been developed, we also implement a means to chain the transformations together using ANT tasks and developed

a web-based Graphical User Interface using PHP [29]. These developments along with additional reasons for creating a plurality of transformations and the details of each will be further explored in Chapter 5.

Chapter 5

Development of the Process Using the Case Study

Using the process defined in Chapter 4, along with the technologies and tools selected, we now endeavor to describe the process by which we developed the model transformation process and user interface tool.

Using the general transformation shown in Figure 2.3 as a guide, we can see that four primary components will need to be developed for each of our ATL transformations shown in Figure 4.8:

1. Source Meta-model
   - Details of how we achieved this step are discussed in Section 5.1
   - Two ECORE files were developed for this
     (a) Simulink Meta-Model
     (b) Intermediate Meta-Model
2. Source Model
   - The processes we used to create these components are detailed in Section 5.2
   - One XMI file was produced using XSLT the initial ATL source model
   - Each additional Source Model was the Target Model of a previous transformation
3. Target Meta-model
   - For most of our transformations, the Target Meta-model was the Intermediate Meta-Model described in Section 5.1.2
   - The meta-model for the SysML Target is discussed in Section 5.1.3

- One ECORE reference was used for the last Target Meta-model

4. ATL Rules files

    - The methodology for creating these components is discussed in Section 5.4.
    - Multiple sets of ATL files were developed for this part of the process

        (a) Simulink Defaults Extraction

        (b) Block Information Creation

        (c) Port Addition

        (d) Port Connection

        (e) Intermediate Model Compression

        (f) Papyrus SysML File Creation

For each of the XSLT transformations used in our process, only the components of Source Model and XSLT Rules are necessary. As noted in Section 4.3, the XSLT does not require explicitly defined meta-models.

5.1 Meta-Models

The first task we undertake in the development of this transformation proces is to create meta-models to represent the structure of the source Simulink model, Intermediate models, and SysML models used in the transformation. As will be described in more detail in Section 5.2, we needed to develop three meta-models:

1. Simulink Meta-model
2. Intermediate Meta-model
3. SysML Meta-model

5.1.1 Simulink Meta-model

The first meta-model we develop is used to define the structure of Simulink models. Due to the limited information available on the general structure of the format of Simulink models, we are required to perform a non-trivial amount of reverse engineering on the example models at our disposal. Particularly, we make use of the structure of the case study model described in Section 4.1.

At the time that we began development of this meta-model, the model format available via Simulink were .mdl files with a text-based format. We assessed the nature of the structure of the case study model and developed a tool for parsing the text of the .mdl file for a model and creating an XML version of the .mdl model that would conform to the meta-model we were developing. As such, much of the early meta-model development and parsing tool development were done concurrently.

Early in the stages of this work, it was discovered that different versions of Simulink would be able to save files in different formats. Beginning with version R2012a, Simulink models could be saved using a new binary format with the SLX extension. The current version of Simulink R2015b uses this format as the default [24]. We were concerned that this new version would limit the work we had already done on creating the meta-model of Simulink to only being useful with older versions of Simulink using the MDL file format. However, the SLX file can be decompressed into a set of files. One of these files, named blockdiagram.xml, contains the Model and Stateflow portions of a model in an XML format.

After investigating the contents of the XML file contained in the SLX formatted model file, it has an almost identical structure to the meta-model we had developed for the MDL format. This gives us much confidence in the understanding we had gained of the structure of a Simulink model. Additionally, only minor changes to to our meta-model are necessary.

The meta-models for Simulink-based models are first developed as UML class diagrams that are then transferred into ECORE models for use in Eclipse. By using mostly Eclipse-based tools for the development of this process, we are able to leverage the ECORE Diagram

tool to graphically depict the model being created [19]. This tool automatically generates the ECORE model that can then be used in our model transformations using ATL.

Our initial version of the meta-model for Simulink do not represent the exact structure of a Simulink model as defined in the SLX format. The reasons for this discrepancy are described in more detail in Section 5.2.

Figure 5.1: Meta-Model of Model Section of SLX Formatted Simulink File

Currently, we have completely defined the meta-model for Simulink Model elements contained within our case study. The representation of the key features of a Simulink SLX formatted model is shown in Figures 5.1 and 5.2. Figure 5.1 shows the meta-model for the Model section. The Model section defines the structure of a Simulink model, which includes

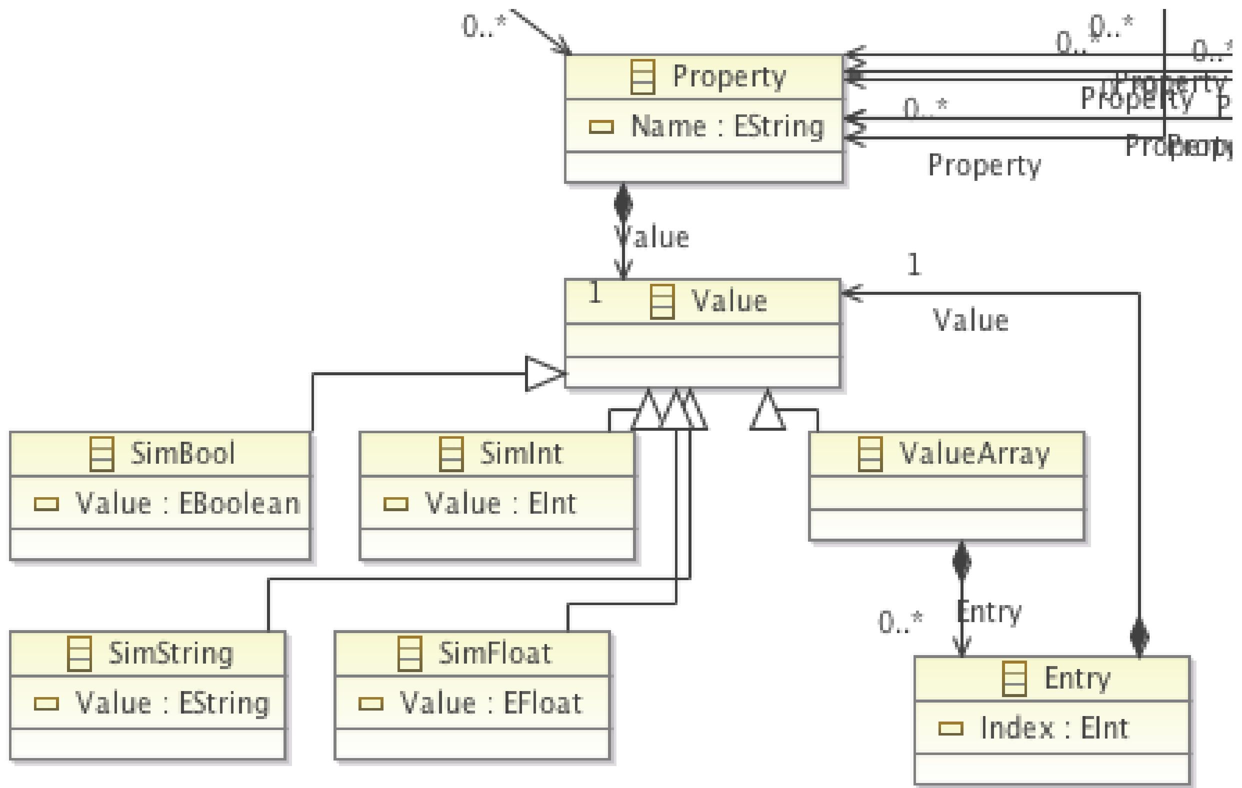

Figure 5.2: Meta-Model of Property Element of SLX Formatted Simulink File

the Blocks and how they are connected. We provide a separate diagram of the Property object in a Simulink model file in Figure 5.2. Early in our process, we made the decision to rely on the XSLT rules portion of our transformation process to do the work of parsing the array elements and rather than the ATL rules. Section 5.2 shows how the work of parsing the arrays for their values and creating the structure in the shown meta-model is accomplished.

The meta-model we developed and used throughout the course of this process is solely intended to represent well the elements contained within the case study model. However, we endeavor to create the meta-model as generic as possible to limit the amount of additional work necessary when expanding this process to include additional, more complex Simulink models. In studying the Block Types not included in the case study, we believe the current meta-model will not require much modification, if any, to accurately describe the structure of the general Simulink model. The work done to investigate the additional Block Types is shown in Section 7.2.

5.1.2 Intermediate Meta-model

Once we had completed the meta-model definition for Simulink models, we develop a meta-model for our intermediate models. This meta-model can be understood to be the following:

- Target Meta-model for
  1. Initial ATL transformation from Simulink
  2. Any Intermediate-to-Intermediate transformations
- Source Meta-model for
  1. Any Intermediate-to-Intermediate transformations
  2. Final ATL transformation to SysML

We desire that this meta-model represent the structure of a model that had all of the necessary information from the Simulink model that would be required for understanding the execution of the model while stripping away anything that is Simulink specific and not needed in other platforms. It is our desire to also have this meta-model be structurally simple so as to be able to have easily defined and understood model elements. We utilize the Composite Design Pattern to simplify the Block-System-Block structure defined in Simulink models [44].

This meta-model can be seen in Figure 5.3. The various components of our meta-model can be understood as the following (from a top-down perspective):

- Model
  - Root element of the model
  - Contains an attribute for the name of the model
  - Has Block elements

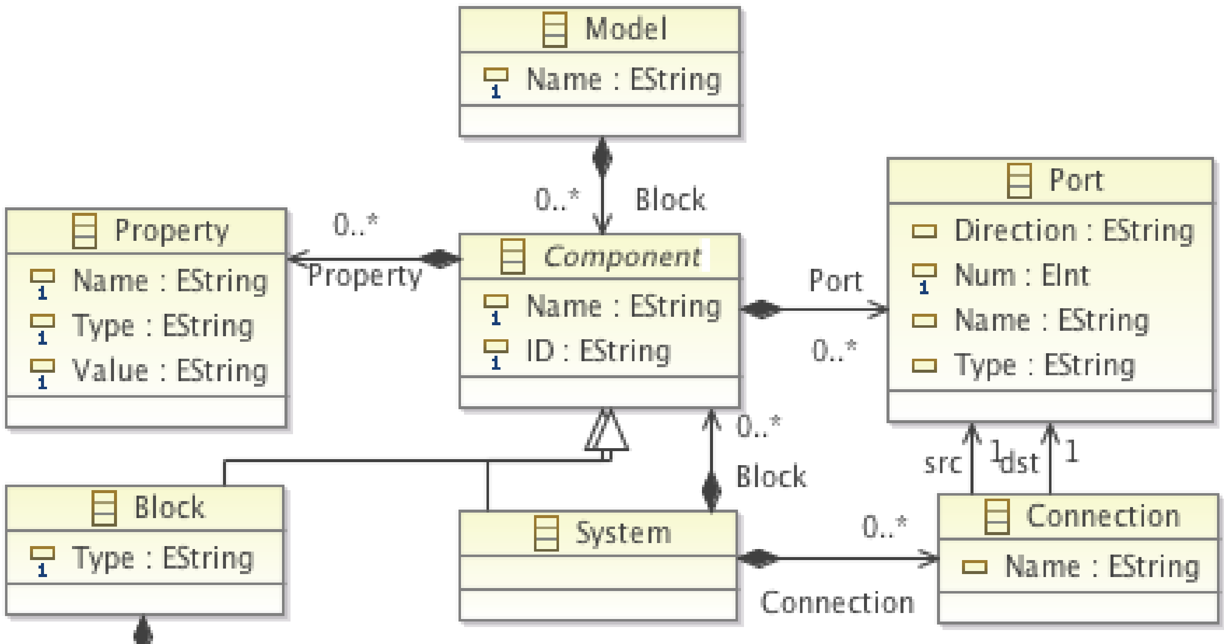

Figure 5.3: Meta-Model of Model Section of SLX Formatted Simulink File

    * Most often is a single System Block
    * Will usually represent the root System of the model

- Component
  - Individual Block elements
  - Core of the Composite Design Pattern
  - Contains an attribute for the name of the component as well as a unique identifier
  - Has Property and Port elements
    * Port elements used to link from one Component to another
  - Block
    * Simple Block that performs a function
    * Type tells what type of function the Block performs
    * Can be the leaf of an S-Function Block that contains MATLAB code
  - System

    * Block that can contain other Blocks
    * Contains the Connection elements that define how all of its subBlocks are connected

- Property
  - Storage for important information on a Block or System
  - Has a Name attribute for storing the purpose of the data
  - Type represents the determined data type of the Value
  - Value has the actual Value of the data
    * Can be a simple data type such as integer or string
    * If an array, will be stored as list of bracketed values
- Port
  - Information about the Port on a Block
  - Direction currently can only be "in" or "out"[1]
  - Num is the Port number and can be from 1 to n with n being the number of Ports on a Block
  - Name used to identify the Port on SysML diagrams
  - Type represents the determined data type of the Port
- Connection
  - A simple element to indicate that a connector exists from one Port to another
  - Has one source and one destination Port; both required
  - Has a Name attribute to identify the Connection in SysML

[1] Simulink only allows for Inport and Outport. However, FlowPorts in SysML can be "in", "out", or "in/out".

As will be further discussed in Section 5.4, our individual intermediate transformations will build the final intermediate model using this "top-down" perspective by first transforming the Model and Components with their Properties, then transforming or creating the Ports, and finally, connecting them via Connections.

5.1.3 SysML Meta-model

For Papyrus, the tool developed for creating UML and SysML diagrams in Eclipse, the meta-model used to define SysML elements is an existing ECORE meta-model available via URL [18] [10] [9]. Even when not connected to the internet, this URL serves as a link to a locally stored meta-model that can be used for either transformations or the viewing of UML and SysML models using Papyrus. Therefore, we are able to use this meta-model as the sole meta-model for transformations to or from SysML. Some of the components of the SysML ECORE meta-model and their UML and Simulink equivalents are given in Table 5.1.

UML Entity	SysML Entity	Simulink Entity
Package	Package	Model
Class	Block	Block
Port	Flow Port	Block (Type Inport) or Port
Connector	Connector	Line
Property	Property	P
Operation	Operation	Function Name
Activity	Activity	Block Functionality
Association	Association	Blocks part of a System

Table 5.1: Equivalent Objects in UML, SysML, and Simulink

5.2 Simulink Model

5.2.1 Simulink Structure

In order to use ATL, the input model to a transformation must be in an XMI format. This necessitates the use of a conversion utility to create the input model from the existing

XML-formatted model from the Simulink SLX model. The model created from this process must conform to the meta-model described in Section 5.1.

As it was discovered that the Simulink XML file stored values as elements of an XML tag and ATL is not able to easily discern these values, this requires a transformation from Simulink XML to another format rather than a simple extension change. ATL relies on attributes of the XML tag for the values in the Object. Therefore, the primary motivation for using XSLT in this manner is to move the values for Properties from the element domain into an attribute named "Value". Additionally, we use this opportunity to interpret any arrays found in the values of important Property Objects. In this way we can standardize the display of array values found in the model[1]. Additionally, we change Property Objects denoted by the letter "P" in a Simulink model with the XML tag "Property" in our XMI model. This is done to avoid confusion with the reserved tag "P" used in various other markup languages such as HTML.

Using a utility to unzip files, the contents of the SLX file are extracted. This results in a directory named the same as the file name for the SLX file. This directory contains 3 subdirectories named "_rels", "metadata", and "simulink". The metadata directory contains an XML file with data about the model, such as the version of Simulink used to develop the model and the author of the model. This data may be useful eventually, but the item of interest for our transformation process is in the simulink directory. This directory contains a file named "blockdiagram.xml". This XML file contains the Model and Stateflow definitions of the Simulink model.

An example of the block structure for a SubSystem Block as it appears in the blockdiagram.xml file of the Team 1 Strategy model file[2] is given in Listing C.1. The line structure detailing how these blocks are connected[2] is given in Listing C.2.

Using the XSLT transformation described in Section 5.5.2, this XML representation is translated into the block structure and line connections shown in Listings C.3 and C.4,

[1]In Simulink the array "[1 2 3 4]" is the same as "[1, 2, 3, 4]"
[2]For the sake of space in this document, some GUI "P" elements were removed from the source XML file.

respectively. It should be noted that in the case of Blocks of type "Inport" or "Outport", the pertinent information is the Port number. This is denoted by the "P" element with Name of "Port". Some blocks in the example shown do not have this element. This is due to the use of default values sections near the beginning of a Simulink model definition (the default Port number is typically 1). We endeavor to keep the structure as similar to the original Simulink model as possible. The two block types mentioned should not be viewed as the only types governed by the use of default values. In fact, all block types present in a Simulink model have a corresponding default values section in the header section of their respective model definition.

5.2.2 Simulink Settings

In addition to the model elements that were desired for the eventual SysML model, the Simulink blockdiagram XML file contains a header with Simulink-specific Properties. Rather than discard these completely, our system retains these Properties in a separate "Simulink Settings" file that can then be used upon a reverse transformation from a SysML model to Simulink model. This model is a useful location to store Stateflow information for a given model, such as the MATLAB code for an S-Function Block.

Since this model's information would be "copy-pasted" into a resulting Simulink model file, it is not necessary to convert its elements, such as Property values into attribute values. We simply need to extract the XML tags and values from the original blockdiagram XML file. This model could be described as the remainder from the original blockdiagram XML model once the information used for the XMI model had been extracted.

5.3 Intermediate Models

As has been previously discussed, a single ATL transformation from a Simulink-formatted model to a SysML equivalent model is quite difficult, at best, and perhaps even impossible. We began initially to develop a pair of transformations, one to transform from Simulink to

an intermediate model and a second to transform the intermediate model into SysML. Unfortunately, this task is made even more difficult by certain aspects of the nature of Simulink models:

- The use of default values in a separate section
- The lack of explicitly defined Port objects
- The use of formatted strings to connect ports
- The use of Branch objects when an output from one Block is connected to multiple inputs
- The addition of Blocks to S-Function Blocks that should not appear in a resulting SysML diagram

It was decided to create a series of intermediate models to represent the "building" of a final intermediate model by adding Objects and attributes as necessary. We will describe each of these models in this section. The rules to create these models will be addressed in Section 5.4.

In order to better understand this process, Figure 5.4 will be referenced throughout this section. This Figure shows a graphical representation of the XML for the Simulink model Team 1 Strategy. If we compare this to the user interface representation seen in Figure 4.3, we can see a visual representation of the extra substructure that exists for the S-Function Block. Additionally, by observing the SysML IBD in Figure 4.5, we can see that it is not desirable for this additional substructure to be transformed into SysML. In the remainder of this section, we detail the intermediate models to begin with the model depicted in Figure 5.4 and result in the creation of the model seen graphically in Figure 5.5.

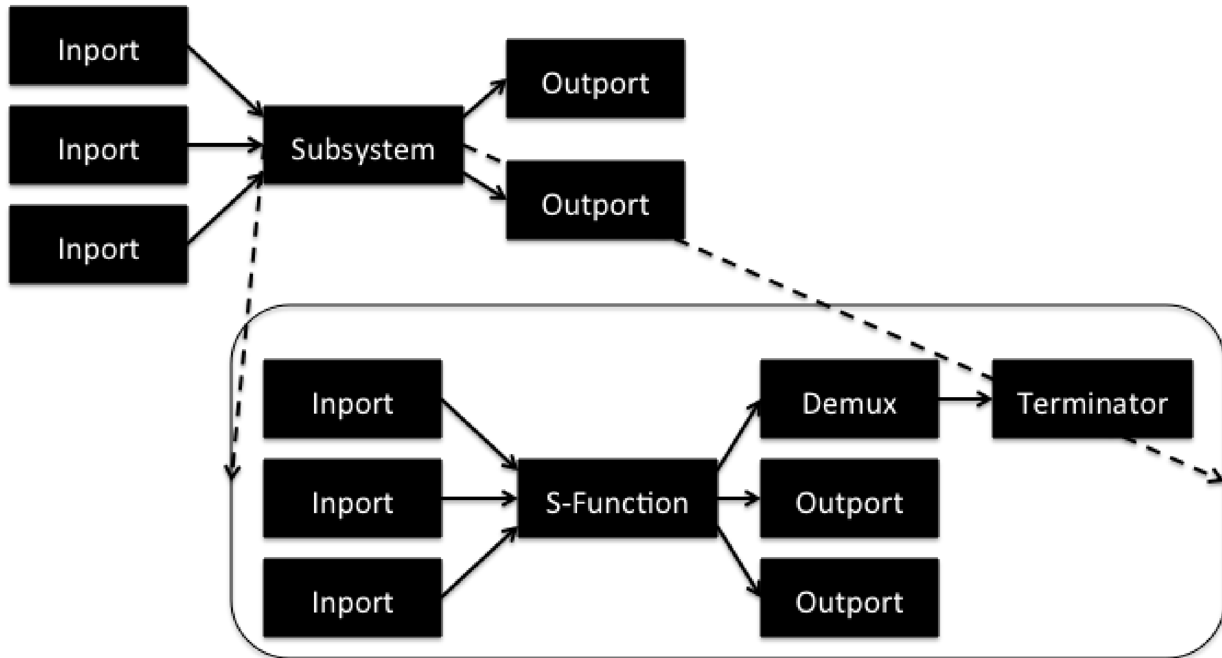

Figure 5.4: Simulink Structure for the Model Team 1 Strategy

5.3.1 Default Property Values

This transformation is the most trivial of all the transformations developed for this process. This model consists solely of a Model Object with a list of Blocks and their Properties. These Properties represent the default values used in a Simulink model when the specific Property does not exist in the model Object. For example, when an Inport Block is encountered in a Simulink model, if it has a Property named Port, this value is used for the port number. Otherwise, it will search for the default value defined in the header of the Model. The structure for the default Property values model can be seen in Figure 5.6.

5.3.2 Components with Properties

The complete list of Properties for a Block can be seen to be the list that is explicitly stated in the Block's Property list combined with the values that appear in the default list. Of course if a value appears in both places, the understanding is that the value explicitly

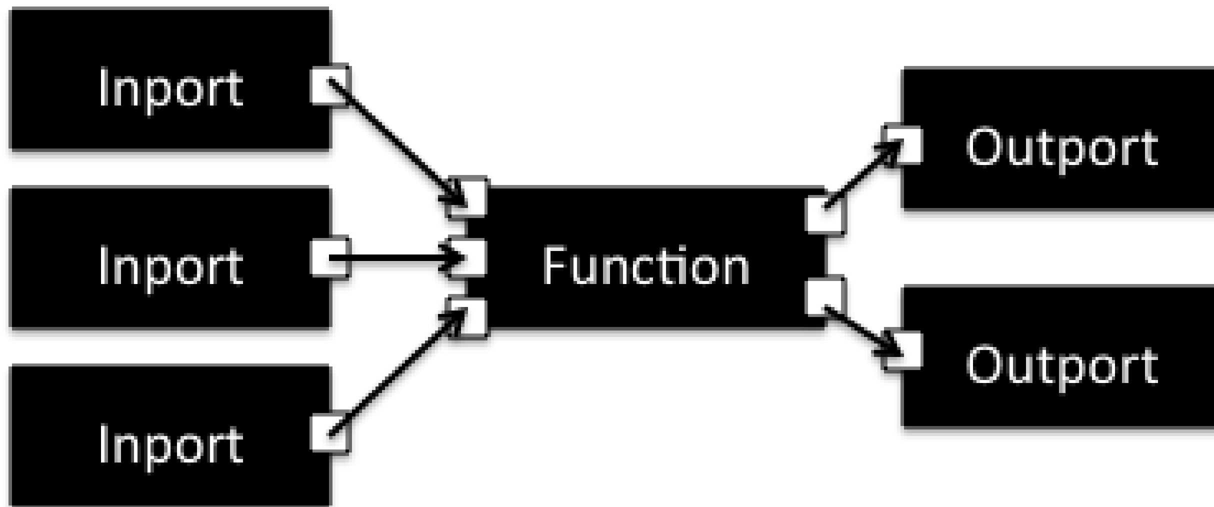

Figure 5.5: Model Used as Source for Final ATL Transformation to SysML

stated overrides the default value. Structurally, the model as a result of this transformation can be seen in Figure 5.7.

As can be seen, only the Blocks and Systems have been transformed at this point. The SubSystem Block with its System Object have been transformed into a single Component of type System. It now contains the transformed Objects that were inside of the Simulink System.

5.3.3 Adding Ports

Now that we have all necessary Properties for all Blocks, we can add the Ports that will be needed to each Block. Generally, Simulink establishes the Port make-up of a Block through the use of the Block's Properties. This is the reason that we must gain all Properties for a Block before we can add its Ports. The structure of the Intermediate model for Team 1 Strategy after we added its Blocks' Ports can be seen in Figure 5.8. Note that the S-Function Block inside of the System Component has three Outports, but the System Block, like the Simulink GUI S-Function Block, has only two. This is a graphical representation of what was discussed in Section 4.2.

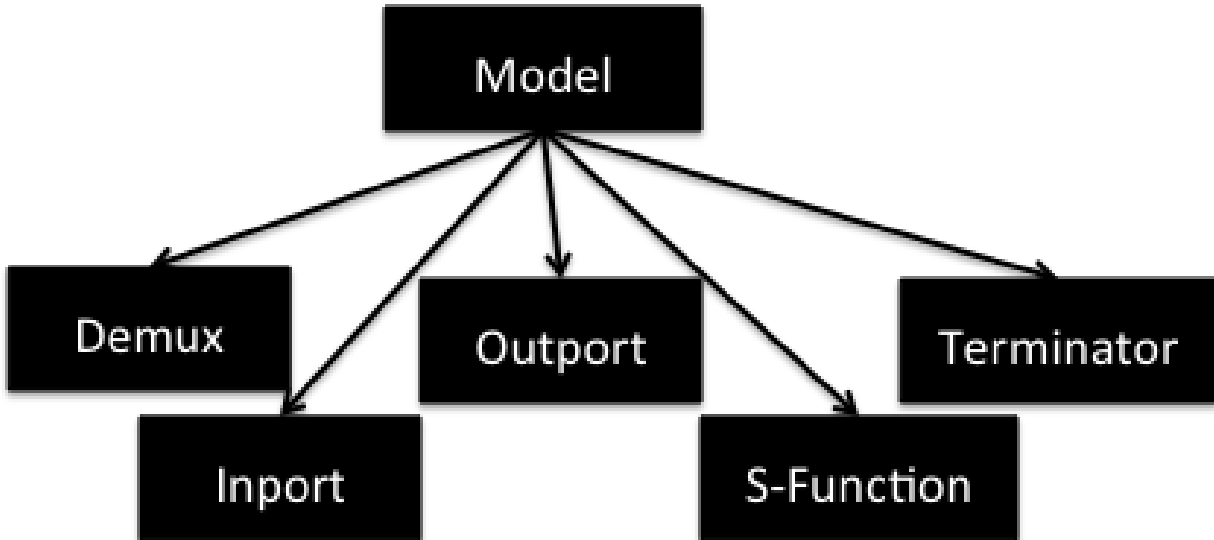

Figure 5.6: Block Default Properties Structure

5.3.4 Connecting Ports

To complete the Intermediate model, we must connect the Ports that were added in the previous transformation. We can use the Line elements in the original Simulink model to determine the appropriate Block and Port to connect as the source and destination of a Connection Object. Figure 5.9 shows this model with all of the Connections. For this graphic, it is understood that a Connection has as its source Port the arrow tail and its destination is the arrow head.

5.3.5 Removing Superfluous Objects

In studying the XML of our case study models, there exists a general structure for S-Function Blocks as described in Section 4.2. Therefore, our final transformation resulting in a model conforming to the Intermediate Meta-model results in the removal of Objects that should not be further transformed into SysML elements, such as the Demux and Terminator Blocks inside of the S-Function System. The rules developed to create this model are described in Section 5.4.5. The model resulting from this transformation was shown earlier in this section. It can be seen in Figure 5.5.

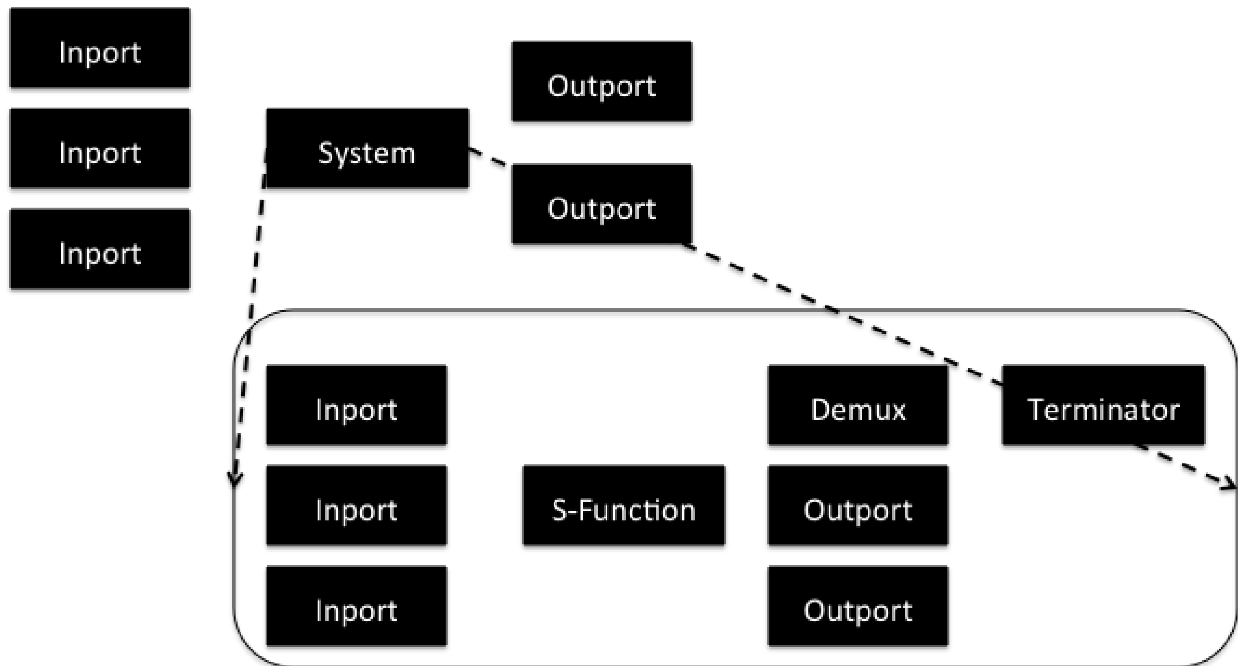

Figure 5.7: Main Structure of Team 1 Strategy

5.3.6 Non-viewable SysML

The final ATL transformation results in a model in which all the pertinent components from the original Simulink model are now represented as equivalent SysML Objects. This transformation results in the creation of two files that will be used to generate the final viewable SysML model:

1. UML file which contains the following information

   - Package Objects
   - Class (Block) Objects
   - Port Objects
   - Connector Objects with "end" children for which Ports are connected
   - Association Objects with how parent-child relationships are defined between Blocks
   - Attribute (Property) Objects

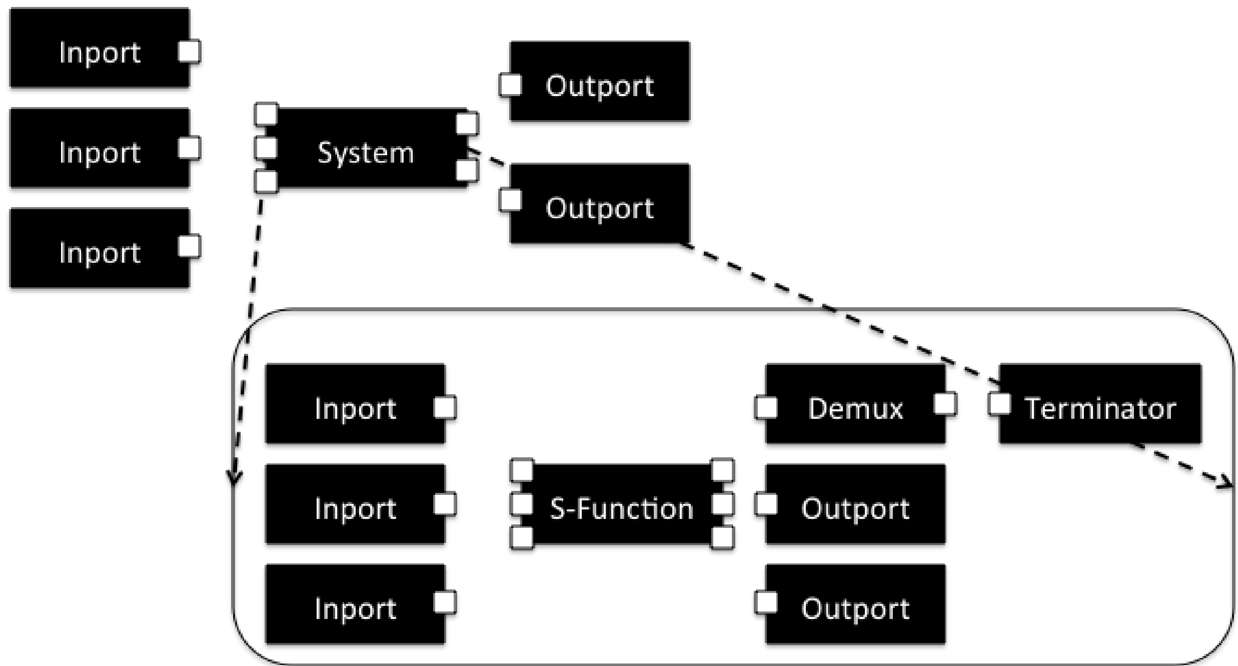

Figure 5.8: Intermediate Model after Adding Ports

2. Notation file which contains the following information

    - Diagram Objects with Type of Diagram
    - Shape Objects referencing most Objects in the UML file
    - Coordinate information for location and size of Objects
    - Edge Objects referencing Connector and Association Objects in the UML file

The final file necessary to make the resulting SysML model viewable using Papyrus is the DI file. This file is relatively simple in nature as it simply contains references to the Diagram Objects in the Notation file that are to be displayed. The process for creating this file will be discussed in Section 6.1.1.

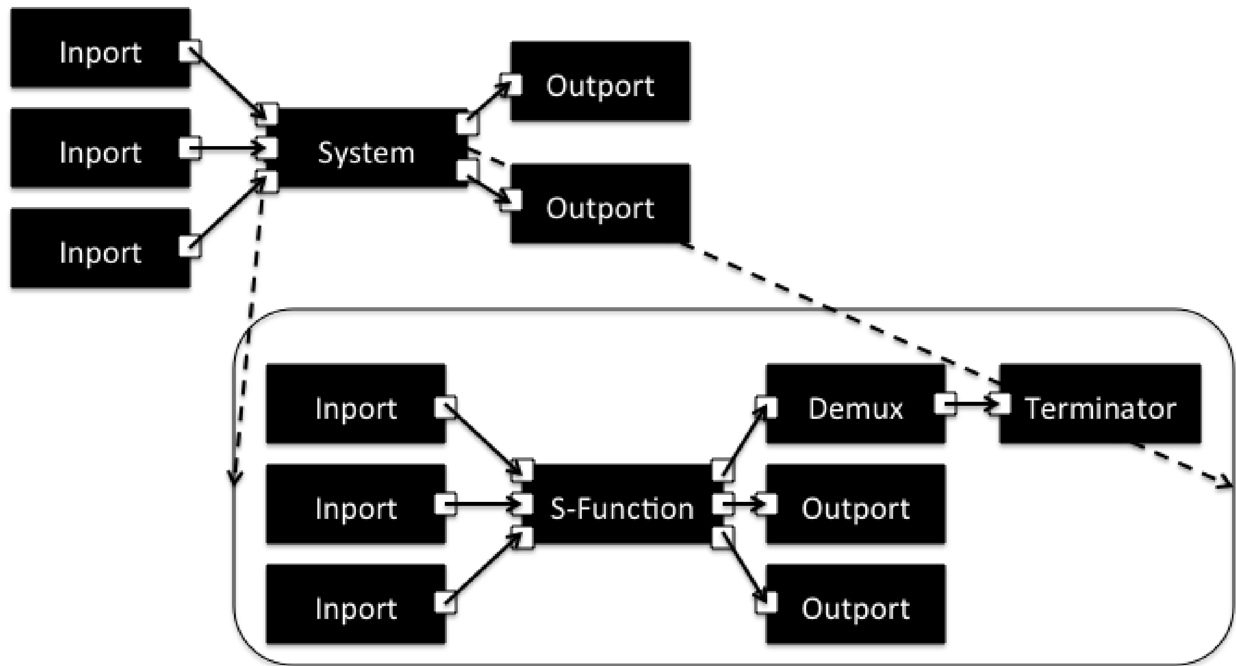

Figure 5.9: Intermediate Model after Connecting Ports

5.4 ATL Rules

In this section, we present the sets of ATL rules that are used to produce the models described in this chapter. Where appropriate, we give an example rule or helper to better understand the transformation being discussed.

5.4.1 Default Property Values

To produce the model described in Section 5.3.1, we create a set of ATL rules to extract the defaults section from the Simulink-based XMI model. As previously stated, this is the most trivial of the transformations developed.

5.4.2 Components with Properties

The model shown in Section 5.3.2 is created using less trivial means than the process of the previous section. This transformation takes two input models, the original Simulink-based XMI model and the Block Defaults model created by the previous transformation.

One output model is created that combines all of the Blocks found in the Simulink model with the Properties as appropriate. An example rule from this transformation can be found in Listing A.1. In this example, we make use of a helper to tell if the Block were a SubSystem and if it had explicitly defined Ports. The point of interest for this rule, though can be found in lines 11 through 14. Here, we determine the Properties to add to the Block. We take all the Properties from the defaults section that have a Type that is the same as this Block Type. If the Property does not exist in the Block's defined Property list already, we add it by using the lazy rule defaultProperty. Lazy rules were necessary when the same Object from a source model, in this case a default Property, would be needed multiple times in a target model. For example, two different Inport Blocks would need the Port number 1 if they were sub-Blocks of different parent System Blocks.

5.4.3 Adding Ports

The next step in building our Intermediate model is to add the necessary Ports to the Blocks that were transformed in the previous step. Unfortunately, Simulink does not explicitly define most of the Ports for Blocks. Instead, we rely on the use of Properties that were determined to contain the Port make up information. For example, if the Property "Ports" is found in a Block's Property list, we can assess that the value of this Property is a two-element array with the first element being the number of Input ports and the second being the number of output ports. If the array has only one element, it is understood to be the number of input Ports. Other Blocks, such as a BusCreator Block, contain a Property for the "Inputs" with an integer representing the number of inputs to the Block while some have a Property for "Outputs". Finally, some Blocks have an assumed Port make-up. The Terminator Block, for example, only has one input Port.

Once we are able to gain the number of Ports necessary for a Block, we can create the Ports and assign them to the appropriate Block. Unfortunately, since there is not an original Port to be transformed, we can not create an ATL rule. In our first attempt at performing

this transformation, we relied on an XSLT transformation rule set to create the Port objects through recursive calling of an XSLT rule. As will be further discussed in Section 6.2.1, we were forced to develop a means to perform this transformation using ATL. Fortunately, an ATL Sequence can be used to create Objects using a lazy rule. Additionally, a helper can be called recursively to generate a Sequence of integers from 1 to n. The helper used to generate the Sequence, seen in Listing A.2, can be used in conjunction with a lazy rule to create a Port, seen in Listing A.3 as seen in Listing A.4. The Block transformation rule creates Ports by obtaining the number of Ports of a certain type, generating a Sequence of integers from 1 to the number of Ports, then collecting the results of performing the lazy rule for creating Ports on each integer value.

5.4.4 Connecting Ports

Now that we have Ports on Blocks, we can continue our process by transforming Lines and Branches from the Simulink model into Connections in our Intermediate model. We begin this transformation by simply performing a 1-to-1 transformation on everything that had been transformed thus far; Blocks, Properties and Ports. This transformation requires the input of two models, the original Simulink-based XMI model and the resulting output from the previous transformation.

In this transformation, when a Line or Branch is encountered, we use a pair of lazy rules to create the resulting Connection Objects. These lazy rules can be seen in Listings A.5 and A.6. Fortunately, all Connections exist in the same System as the original Line or Branch. Unfortunately, not all Lines or Branches consist of a simple structure. In other words, some Lines have a simple single source, single destination pairing. Others, however, have a single source, but multiple destinations through Branch Objects. However, the Line Object always contains the source information. Also, if a Branch has no sub-Branches, then it contains the destination information. As such, the development of the rule to create the Connections for a System becomes the rule seen in Listing A.7

5.4.5 Removing Superfluous Objects

As stated above, the structure of the S-Function Block in a Simulink model includes Objects not viewable in the GUI while also being undesirable in the final version of a generated SysML model. As a result we create a set of ATL rules explicitly to search for this general structure defined in Section 4.2. Because this sub-structure is not limited to a specific number of inports or outports on an S-Function Block, we need to define helpers to search for a generic structure. The rule for creating a Function Block from a System with the sub-structure is given in Listing A.8. As can be seen, this rule uses a helper named "isFunctionBlock" to determine if the sub-structure exists. If it is found, then this rule is used to transform the System into a single Block of type Function. Listings A.9 and A.10 show the ATL code for the helper used to determine if the System is a Function Block. These Listings also show some of the sub-helpers used for this code. The first thing this helper looks for is if the System has Blocks of Type "S-Function", "Demux", and "Terminator" inside of it. If these three do not exist, the helper returns false. Otherwise, it returns a Boolean value that is true only if all of the following criteria is true:

- The number of "in" S-Function Ports equals the number of Inport Blocks in the System
- The number of "out" S-Function Ports equals the number of Outport Blocks in the System plus one
- A Connection exists from the first outport of the S-Function Block to the inport of the Demux Block
- A Connection exists from the Demux Block outport to the Terminator inport
- The number of "in" S-Function Ports is equal to the number of connections to Inport Blocks in the System[1]
- The number of Outport Blocks is equal to the number of connections to "out" S-Function Ports[2]

Additionally, a helper to determine if a model element is a part of a Function is necessary. This helper is shown in Listing A.11 and is used for all Port Objects. If a Port is a child of a Block inside of a Function, then it should not be transformed.

5.4.6 Non-viewable SysML

This set of ATL transformation rules is the most non-trivial of all the rule sets developed. It is necessary to create both files described in Section 5.4.6 concurrently. The reasons discovered for concurrent transformations will be explained more fully in Section 6.4. However, the structure of these two files are drastically different. As an example, if a Block contains another set of Blocks inside of it, the UML file represents this using an Association Object as does the Block Definition Diagram in the Notation file. However, in the Notation file, we also find that the parent Block is an XML parent to the other Blocks inside of it for the Internal Block Diagram. Our ATL rules must be manipulated to create all these structures concurrently.

One of the rules from this transformation is shown as an example in Listings A.12, A.13 and A.14. This code segment shows how much is involved in simply taking a Block from our final Intermediate model and creating the necessary structures for a SysML Block. Much of the tasks in the "do" section of this rule are for the purposes of maintaining the Blocks already transformed and their positions. While we do not intend for our final SysML model to be completely aesthetically pleasing, we understand the necessity to have Blocks not be all in the same exact location. As can also be seen in this Listing, the primary focus of the transformation rule in the from/to sections is to create the UML Objects. The additional tasks of creating the necessary Notation elements is relegated to the do section using called rules such as CreateBDDBlockShape shown in Listings A.15 and A.16.

[1]These Connections are only counted if the port number of the Inport Block matches the port number to which it is connected on the S-Function Block

[2]These Connections are only counted if the port number of the Outport Block is one less than the port number to which it is connected on the S-Function Block

When creating these Notation Shapes, in order to place them appropriately into the correct hierarchy, the best course of action is to insert into the type string, values of the parent and diagram delimited by specified special characters. Once the transformation is complete, this superfluous information can be removed from the model using a simple text replace algorithm. Initially, this task was placed in a final XSLT transformation, but similar to the issue encountered in the transformation described in Section 5.4.3, it was decided that we should develop means to accomplish this in ATL. This is done through a series of calculated decisions in which we force the transformation to order the Objects transformed. In doing this, we manipulate the transformation to process the Model Object last, rather than first. In the "do" section of the Model transformation rule, we place a series of text replace tasks to remove these values from the type string for all SysML Objects.

As stated, this transformation results in the structure and pertinent information for a valid SysML model, but is not viewable due to the absence of a DI file. This file is created through an XSLT transformation described in Section 6.1.1. At the conclusion of that Section, Figures of the final SysML model can be seen.

5.5 XSLT Rules

In this section, we continue our discussion of the transformation rules developed by presenting the sets of XSLT rules that were used to produce the rest of the models described in this chapter. Similar to the previous Section, Where appropriate, we give an example rule to better understand the transformation being discussed.

5.5.1 Simulink Settings

The model described in Section 5.2.2 is created using a relatively trivial XSLT transformation. In fact, the entirety of this rule set is shown in Listing B.1. This transformation transforms any Objects that match a defined list and skips all others. Additionally, it transforms all P elements that are children of these Objects and do not fall into a category that

should be skipped. The skipped P element types are also defined using a list. For each Object transformed, this transformation simply copies the Object as it exists into a new XML document.

5.5.2 Simulink Structure

To produce the model defined in Section 5.2.1, we also make use of XSLT. This transformation is less trivial than the one previously discussed. Similar to the transformation for the Settings model, this transformation uses a defined list of Objects and Property types that should be transformed. As noted previously, this transformation is primarily responsible for interpreting arrays and transforming values into attributes. Arrays are interpreted using a recursive rule so that we are able to accurately create multi-dimensional arrays shown in Listings B.2 and B.3. The rule for creating Entries in the Array is called ProcessArrayPortion and can also be seen in this Listing.

For P elements, we also change the Object name to "Property". This is done to avoid any potential confusion in using the reserved P tag. The most useful parts of this rule are shown in Listing B.4. Some of the specifics of this rule have been removed for readability.

Chapter 6

Validation

6.1 Validation Methodology

Now that our transformation process is complete, we undertake the task of detailing our methodology for validating that our transformation process results in models useful for their intended purposes. This means that models transformed into SysML should be valid SysML-defined models that accurately represent the structure and functionality of the original Simulink model. Additionally, models that result from the transformation process from a well-defined SysML model into a Simulink model should result in an executable model that produces correct results.

6.1.1 Viewable SysML

The final XSLT transformation used in our development process is to produce the DI file necessary to be able to view the SysML models. This transformation requires us to extract the XMI:IDs of the various Diagrams in the SysML Notation file then place those IDs as part of href attribute values in a specific structure. The XSLT rule to obtain all of the XMI:IDs is shown in Listing B.5. As can be seen, we opted to get the IDs in a certain order: BDD, IBD, ACT. By doing this, we ensure this to be the order of display when the diagrams are viewed in Papyrus. This seemed to us to be the most reasonable order for displaying SysML models.

After successfully obtaining the XMI:IDs, we are able to place these IDs into the structure of the DI file as shown in Listing C.5. Now that we have successfully identified the diagrams to be viewed in our SysML model, we can see the final output shown in Figures

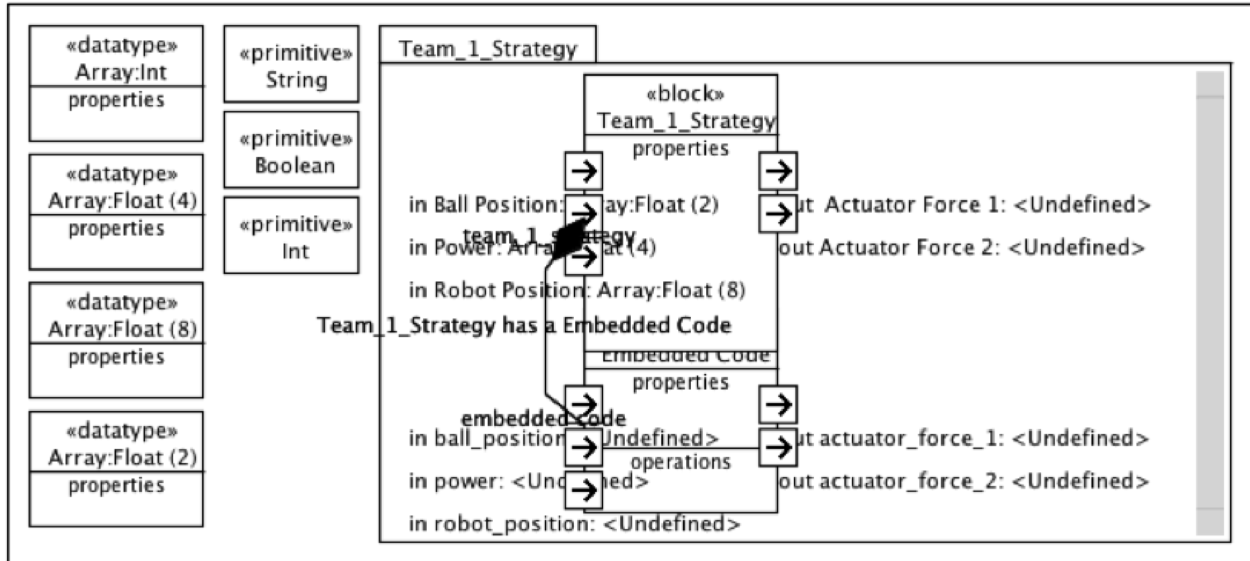

Figure 6.1: Generated BDD for Team 1 Strategy

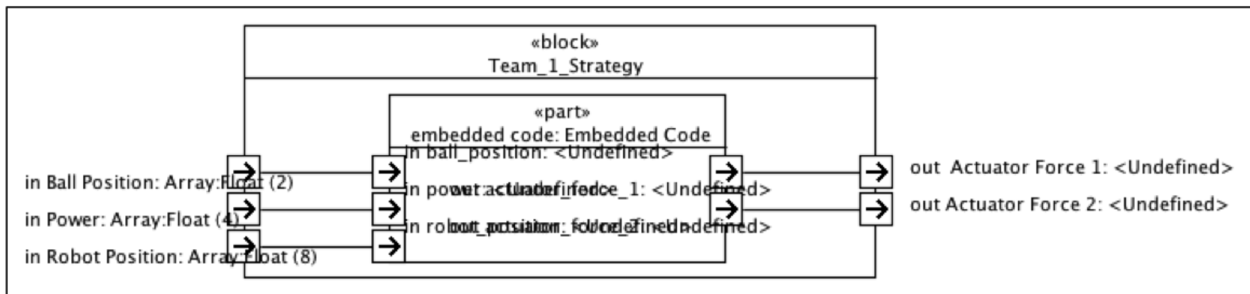

Figure 6.2: Generated IBD for Team 1 Strategy

6.1 and 6.2. Additionally, selected SysML diagrams from the larger RoboSoccer model can be seen in Figures 6.3, 6.4, and 6.5.

6.1.2 Generating a SysML Model

Our first task is to generate a full SysML model from the existing Simulink model files for our case study. Using the user interface defined in the previous section, we perform the transformation for all three model files contained in the case study. The resulting model diagrams can be seen in Figures 6.6 - 6.13. While these models may be unlike those that would be created by a human engineer, our process is mostly concerned with accurately

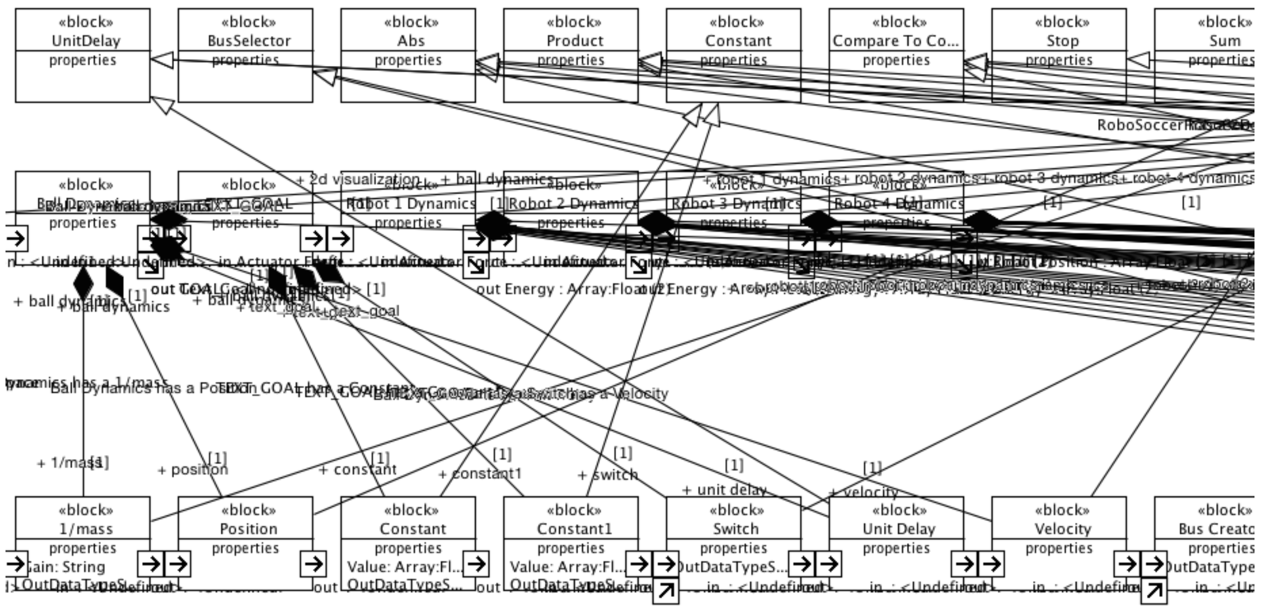

Figure 6.3: Generated BDD for RoboSoccer

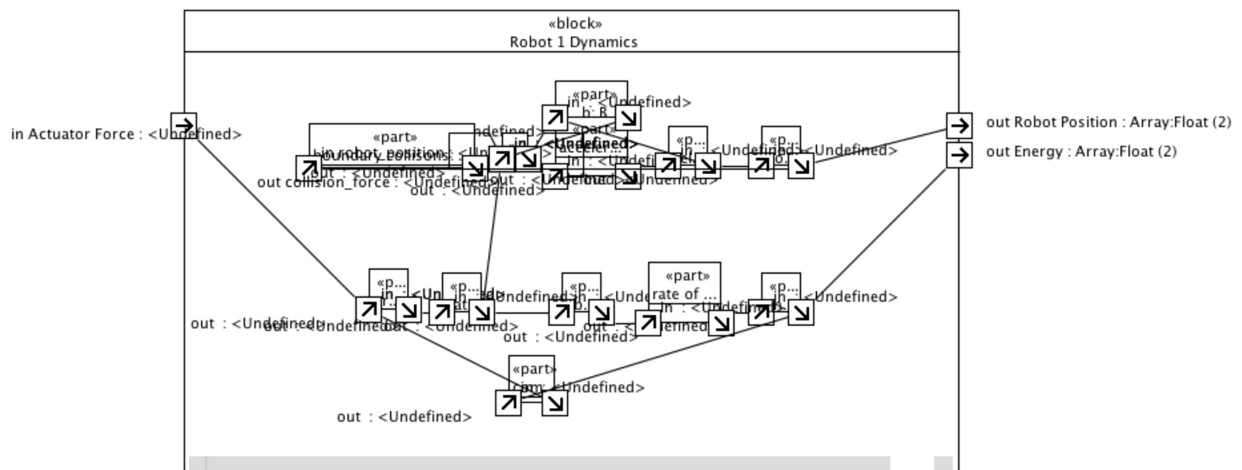

Figure 6.4: Generated IBD for RoboSoccer

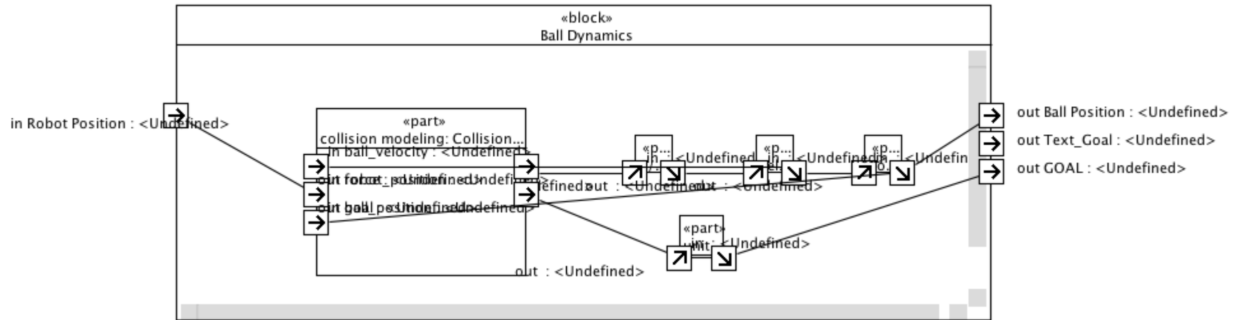

Figure 6.5: Generated IBD for RoboSoccer

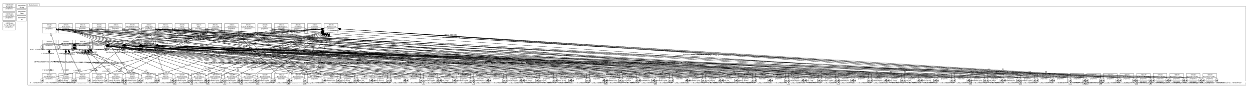

Figure 6.6: SysML BDD for RoboSoccer

representing the contents of the original Simulink model in the .uml file of the SysML model files.

Notably, all blocks from the original Simulink model can be seen in the resulting SysML model. Additionally, they are all connected appropriately. Finally, we endeavor to have the models be as aesthetic as possible. Therefore we achieve the following guidelines for the SysML diagrams:

- BDD
  - The first row of Blocks are the Root model and all Generalized Block Types
  - The second row of Blocks are Systems that are direct children of the root model
  - The third row of Blocks are Systems that are children of other Systems
  - The final row of Blocks are children Blocks and Function Blocks
- IBD
  - We created an IBD for each System in the original Simulink model
  - We interpreted the coordinates and size of the Blocks from the original Simulink Properties

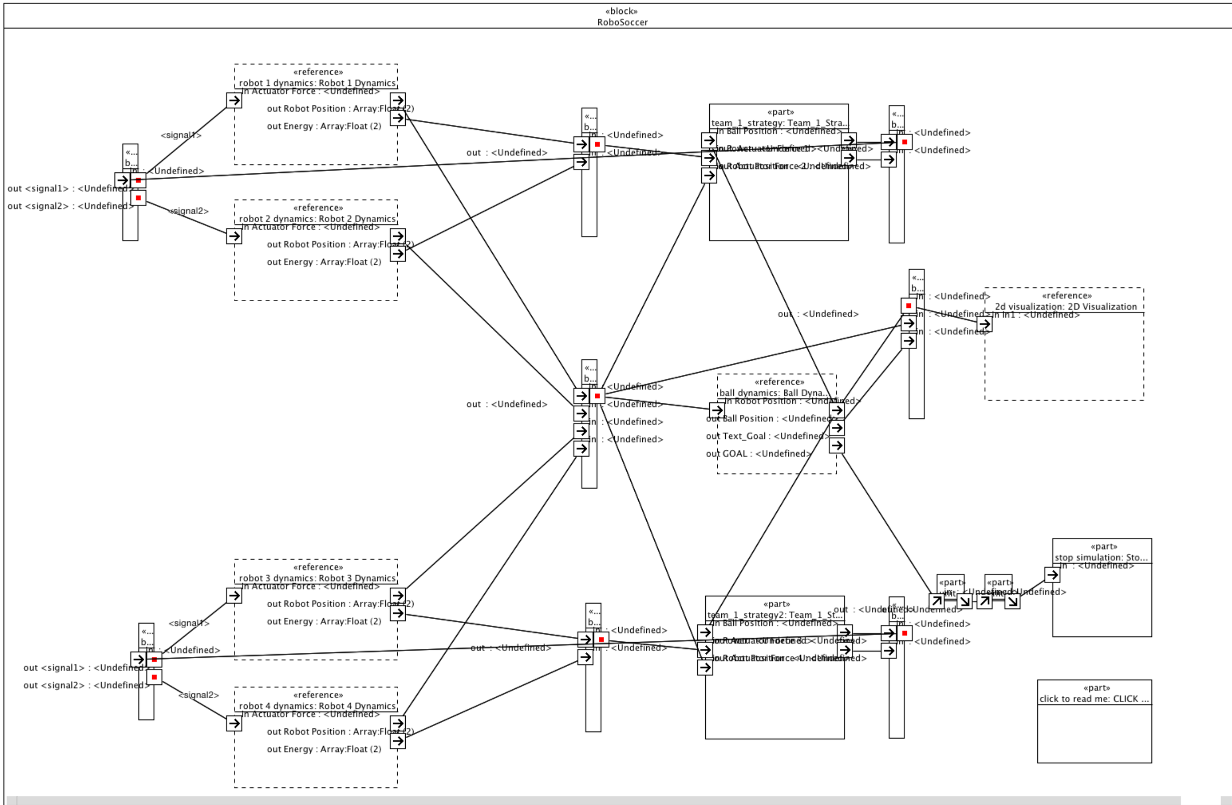

Figure 6.7: SysML IBD for RoboSoccer

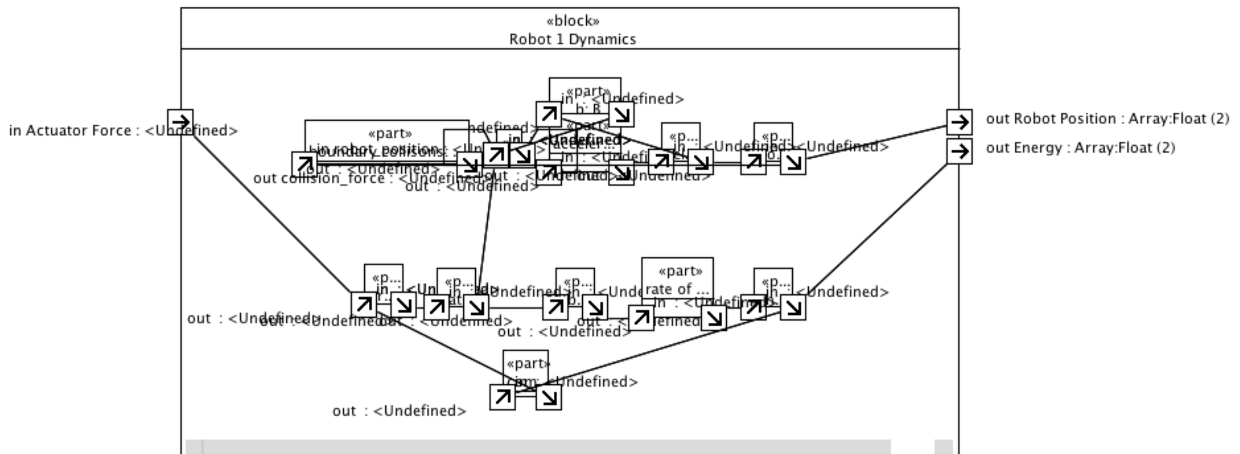

Figure 6.8: SysML IBD for Robot 1 Dynamics

Figure 6.9: SysML IBD for Ball Dynamics

Figure 6.10: SysML BDD for Team 1 Strategy

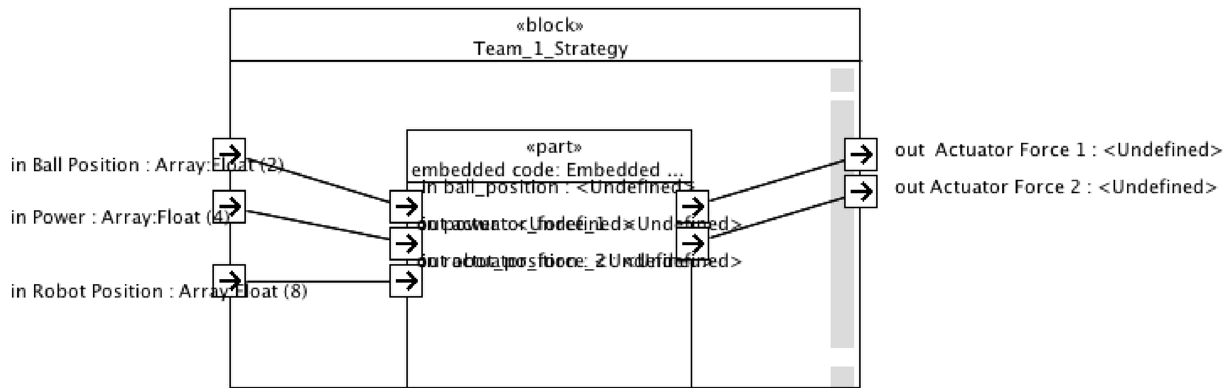

Figure 6.11: SysML IBD for Team 1 Strategy

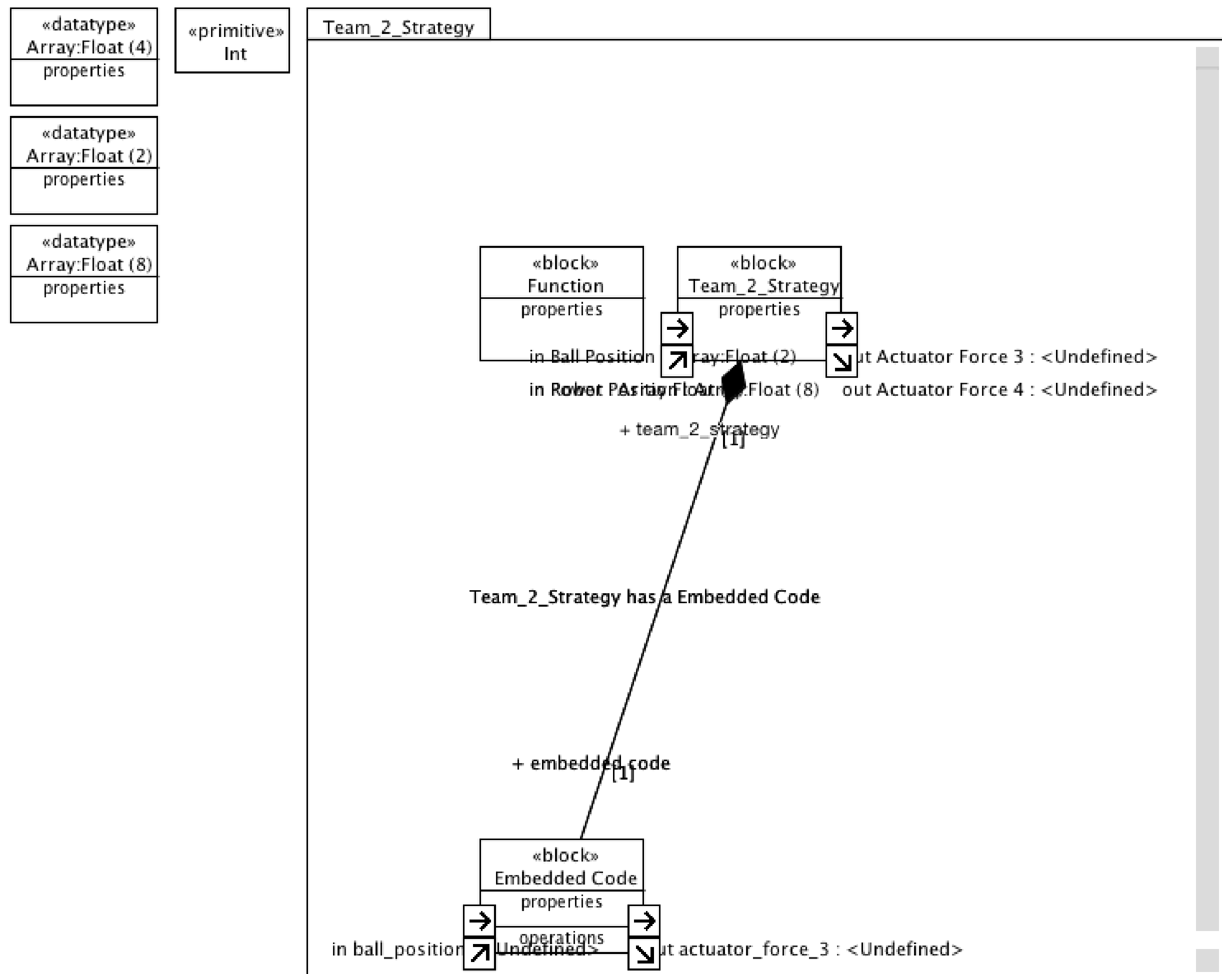

Figure 6.12: SysML BDD for Team 2 Strategy

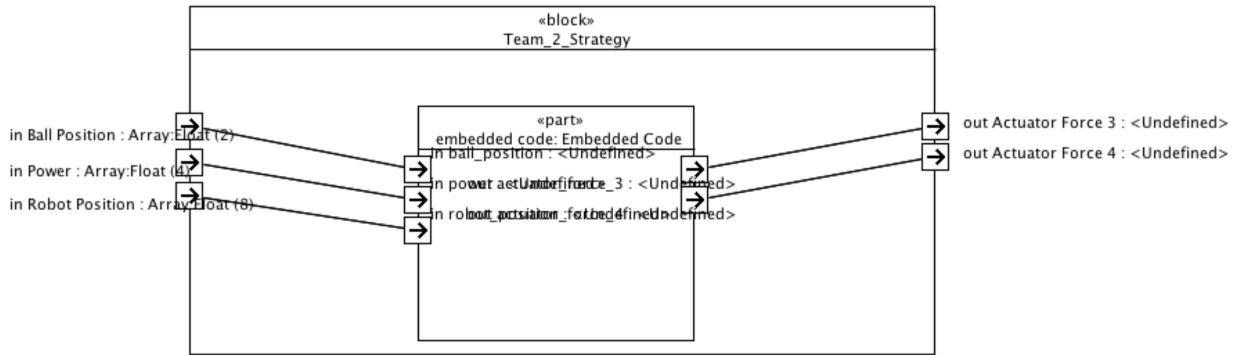

Figure 6.13: SysML IBD for Team 2 Strategy

- ACT
  - We created an ACT for each S-Function Block in the original Simulink model
  - Each ACT is currently empty. As explained in Section 7.2, we will expand to populate these diagrams

6.1.3 Re-creating a Simulink Model

Once we had a SysML representation of the original Simulink model, we perform a "reverse" transformation on this generated SysML model to create a new Simulink model. The Team 1 Strategy model can be seen in Figure 6.14. As can be seen, generated Simulink models are not visually the same as the original models. However, as can be seen in Figure 6.15, when expanded, the Team 1 Strategy model is structurally the same as the original Team 1 Strategy model (Figure 4.3). Again, our process is mostly concerned with maintaining accuracy in structure and execution; not in aesthetics. Ideally, many of the users of our transformation process will not have direct access to the generated executable models. Our process will generate the models, run the simulations, and report the results directly to the users. As can be seen in Figure 6.16, The MATLAB code embedded in this S-Function Block is the same as the original code, due to it being retrieved from the Simulink Settings model created in the first XSLT transformation.

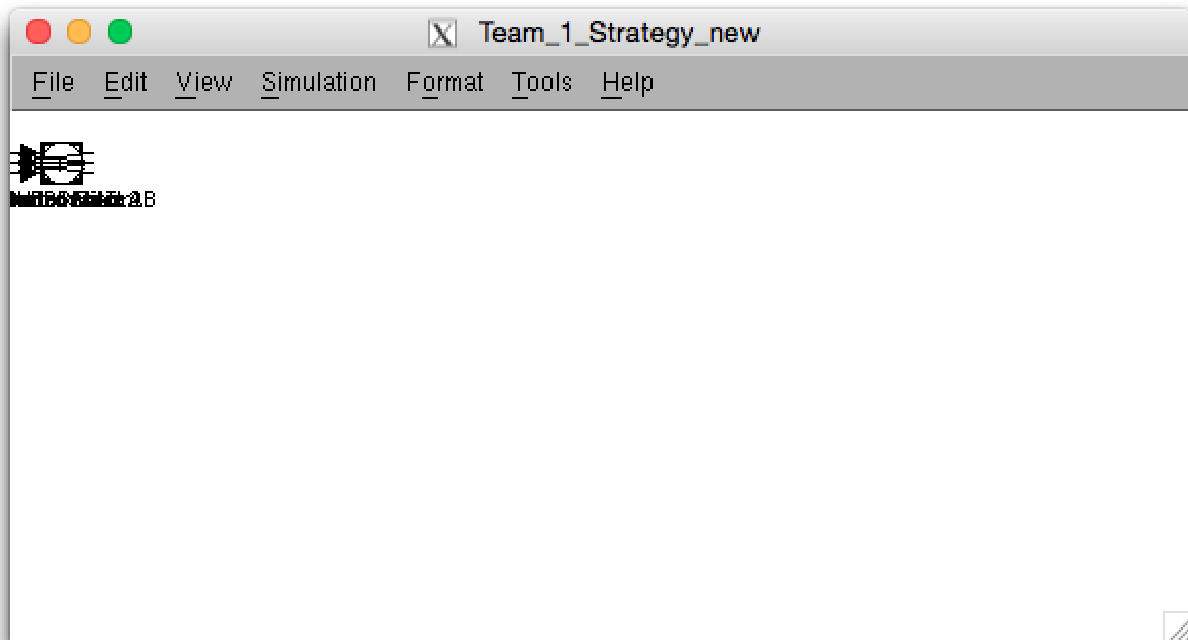

Figure 6.14: Simulink GUI for Generated Team 1 Strategy

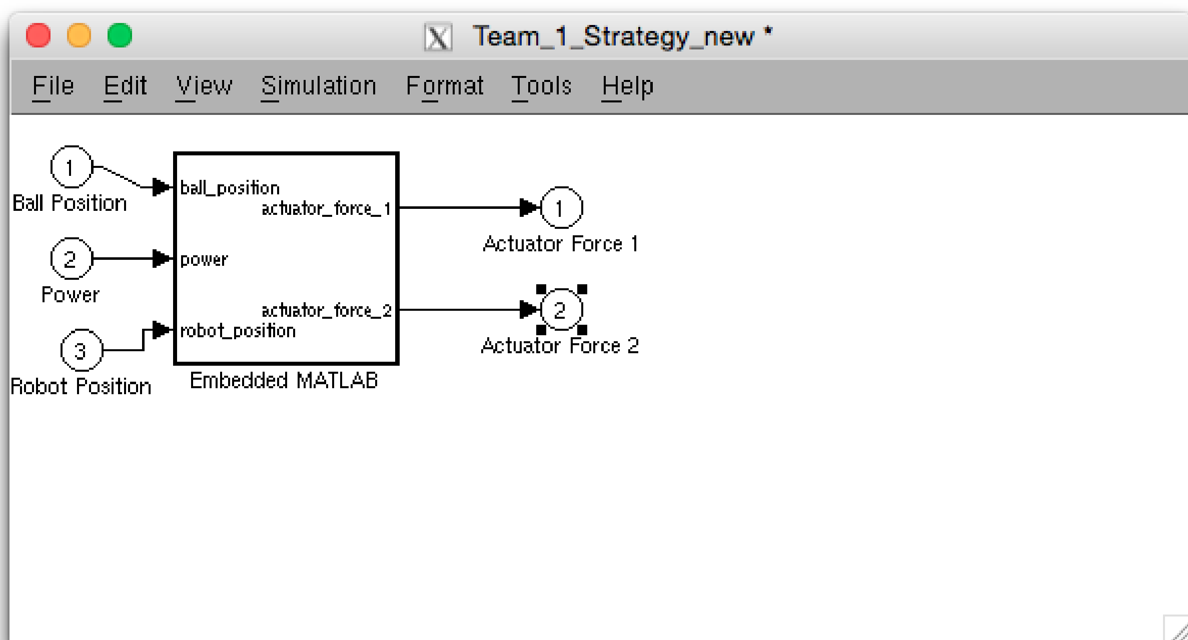

Figure 6.15: Simulink GUI for Generated Team 1 Strategy Expanded

```
function [actuator_force_1, actuator_force_2] = fcn(ball_position, power,  robot_positi
%#eml

d1=(robot_position(1:2,1)-ball_position);
d2=(robot_position(3:4,1)-ball_position);
if dot(d1,d1)>5
    actuator_force_1=-22*d1;
else
    r1=(ball_position- [200;50]);
    actuator_force_1=-11*r1;
end
if dot(d2,d2)>5
    actuator_force_2=-25*d2;
else
    r2=(ball_position-[200;50]);
    actuator_force_2=-15*r2;
end
```

Figure 6.16: Simulink GUI for Generated Team 1 Strategy MATLAB Code

6.1.4 Validating Models

Now that we have a generated model in the Simulink domain that should execute and produce the same results as the original model, we can verify with some degree of confidence that our transformation processes create accurate representations. We do this through a variety of tests:

1. Do the original simulation results match those of the generated model?
2. If a structure or connection is altered in SysML, is it reflected in the generated Simulink model?
3. If a structure of connection is altered in Simulink, is it reflected in SysML as well as the generated Simulink model?

Matching Simulation Results

Our first test of the "round-trip" transformation is to verify that if a simulation is executed using the Simulink model resulting from such a transformation, its results will sufficiently match those of the original model. Fortunately, this is a relatively simple task since the RoboSoccer model does not make use of random number generators. It is a deterministic model where the outcome is entirely dependent on the initial conditions of its parameters. Therefore, the verification process is done by running a number of simulations each time changing the parameters for a robot's stamina. We then execute the same simulations with the same parameters using the transformed model instead of the original. For each simulation, we export the data for the positions of the ball and the four robots to a spreadsheet. We then compare the data to be certain that it is the same for each of the variables. The results of these simulations can be seen in Tables 6.1 and 6.2.

Robot 1 Stamina	Robot 2 Stamina	Simulation Time Steps	Ball Final Position	Scoring Team
0.1	0.1	1486	(205.01, 41.586)	1
0.1	0.5	4128	(201.11, 53.974)	1
0.1	1.0	4324	(200.49, 47.778)	1
0.5	0.1	504	(200.31, 48.796)	1
0.5	0.5	4480	(-0.51145, 58.802)	2
0.5	1.0	3103	(200.7, 57.959)	1
1.0	0.1	1499	(200.38, 43.098)	1
1.0	0.5	1132	(200.25, 48.354)	1
1.0	1.0	2922	(202.99, 57.328)	1

Table 6.1: Results of Simulations Using Original

Robot 1 Stamina	Robot 2 Stamina	Simulation Time Steps	Ball Final Position	Scoring Team
0.1	0.1	1486	(205.01, 41.586)	1
0.1	0.5	4128	(201.11, 53.974)	1
0.1	1.0	4324	(200.49, 47.778)	1
0.5	0.1	504	(200.31, 48.796)	1
0.5	0.5	4480	(-0.51145, 58.802)	2
0.5	1.0	3103	(200.7, 57.959)	1
1.0	0.1	1499	(200.38, 43.098)	1
1.0	0.5	1132	(200.25, 48.354)	1
1.0	1.0	2922	(202.99, 57.328)	1

Table 6.2: Results of Simulations Using Generated Model

Altering SysML Models

The next evidence we seek is to be certain that a change in the SysML model generates an equivalent change in the resulting Simulink generated model. One of the alterations tested is that of changing the structure of connections for the Team 1 Strategy model. As can be seen in Figure 6.17, when we modify the Team 1 Strategy model by switching the output ports of the Function block to the output ports of the main block, the resulting generated Simulink model is shown in Figure 6.18.

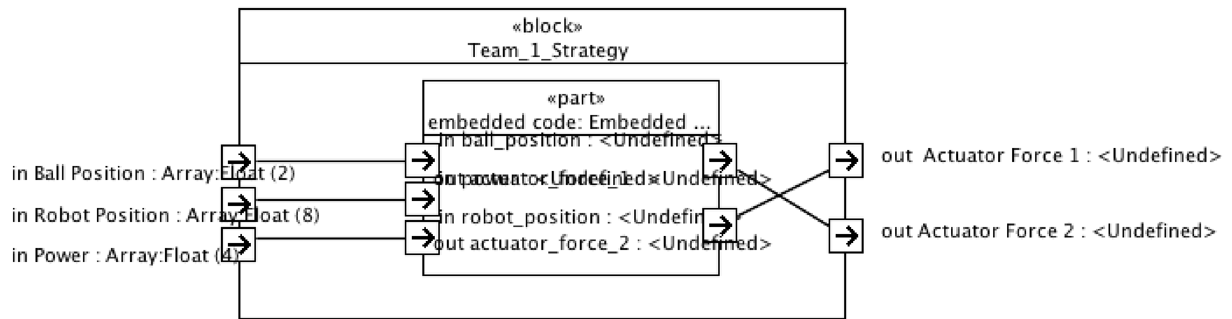

Figure 6.17: SysML for Team 1 Strategy with Modification

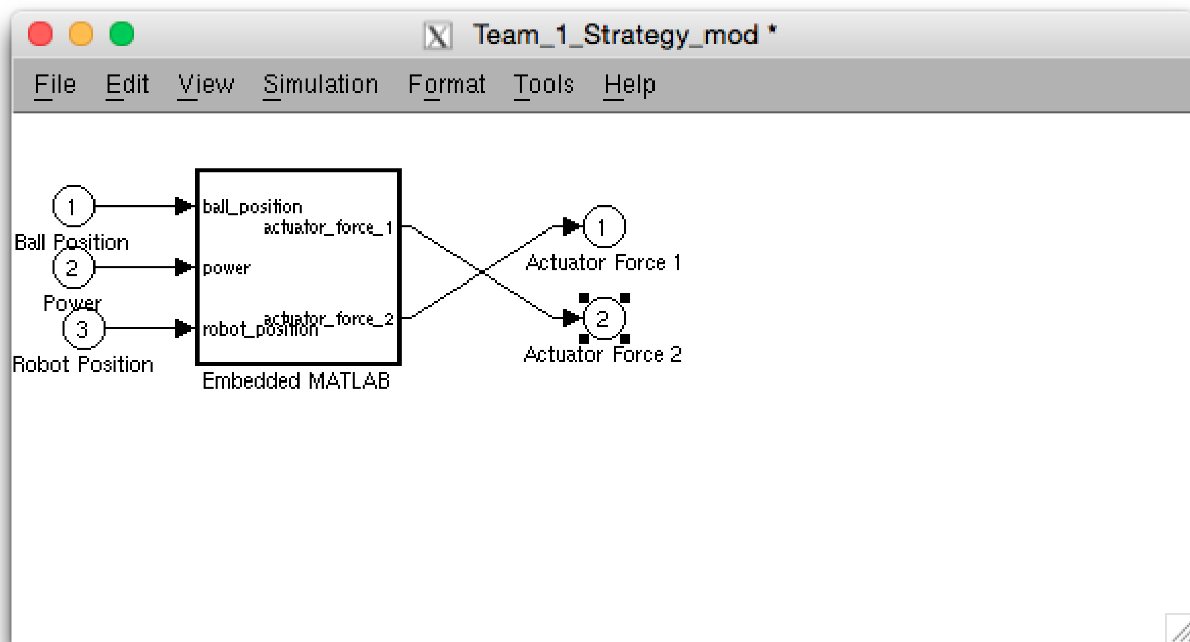

Figure 6.18: Generated Simulink for Team 1 Strategy with Modification

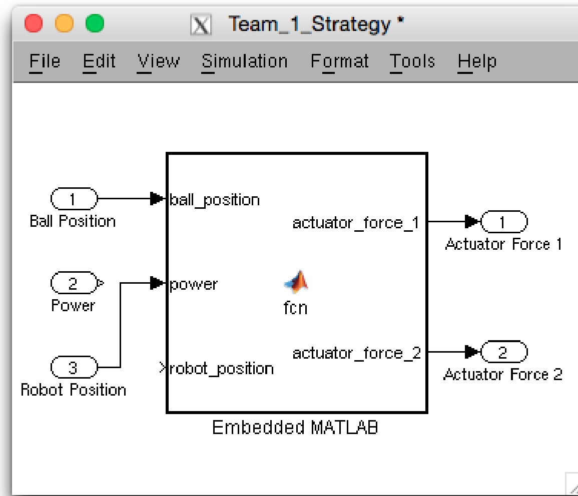

Figure 6.19: Modified Simulink for Team 1 Strategy

Altering Simulink Models

We then take the original Simulink model for Team 1 Strategy and in a similar manner to the previous test, modify the way that the Blocks are interconnected. This time, we remove the Line from the Power Inport Block to the power input of the S-Function. Additionally, we move the Line from the Robot Position Inport Block to the power input of the S-Function Block. This model can be seen in Figure 6.19.

While this model would likely result in many errors if an execution were attempted, we do this only to test if the structural features of this model are maintained through the two transformation processes. The resulting SysML model and generated Simulink model can be seen in Figures 6.20 and 6.21, respectively.

6.2 User Interface

To make our transformation process useful to those interested in performing such transformations, we also provide a user interface for facilitating the automated transformation of models.

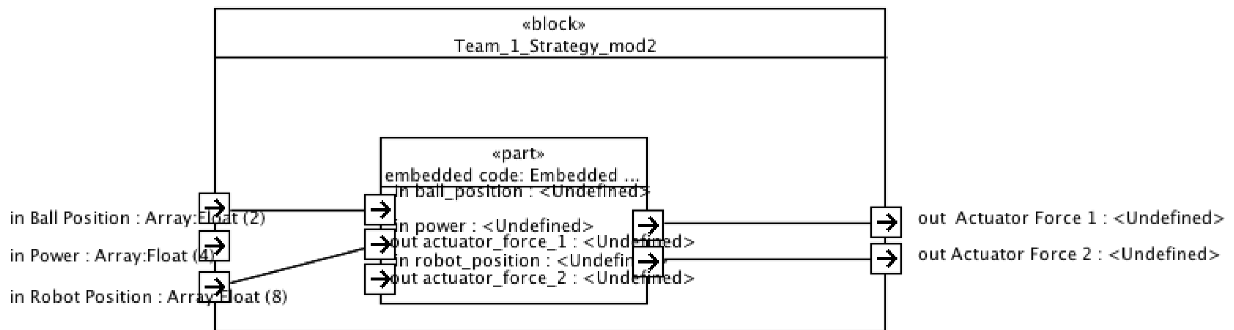

Figure 6.20: Modified Simulink for Team 1 Strategy Transformed to SysML

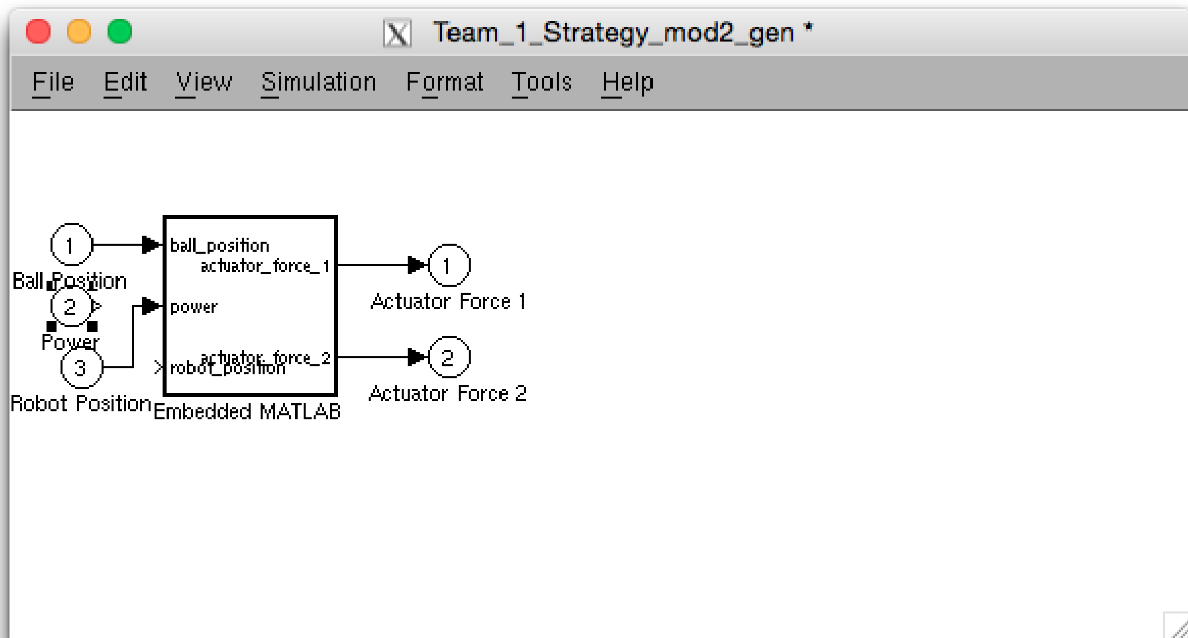

Figure 6.21: Modified Simulink for Team 1 Strategy to SysML to Simulink

6.2.1 Chaining Transformations

In order to perform a transformation using the user interface that would eventually be developed, we first deemed it necessary to be able to chain transformations together using the output of one intermediate transformation as the input to a subsequent transformation. We find that this can be relatively easily accomplished using ANT tasks in Eclipse [1]. The process for incorporating various tasks in a list of ANT tasks is as follows:

- unzip task [2]
  - Used for extracting the contents of a compressed file
  - Was used for decompressing the SLX model file to obtain the blockdiagram.xml file
- zip task [4]
  - Used for compressing a folder into a single file
  - Was used for compressing the contents of a Simulink model into an SLX file in the "reverse" transformation
- xslt task [3]
  - Used for performing an XSLT transformation
  - Takes as attributes values for the location of the XSLT rules file, an input model, and an output model
  - Saves the output model as a new file on the computer or server
- atl tasks [13]
  - loadModel
    - Loads a model file from the computer or server
    - Source models and meta-models must be loaded this way

    - Must give a meta-model name
    - If loading a source model, meta-model must be previously loaded
  - launch
    - Uses a path to a compiled set of ATL rules
    - Uses an input model that was loaded through loadModel
    - Generates an output model as a result of the transformation
    - Does not save the model; only stores the output model in a temporary location
  - saveModel
    - Takes a temporary model stored and saves it to the computer or server

These ANT tasks are used to chain the transformations together as shown in Figure 4.8. Example ANT tasks are shown in Listing D.1. The use of ANT tasks led to the alteration of some of our XSLT transformations into ATL transformations. These design changes are discussed in Section 6.4.

6.3 Graphical User Interface

Once we had the ability to chain the transformations using ANT tasks, we are able to begin the development of a GUI for performing the transformations. We make use of an Eclipse Headless Build to facilitate the execution of an Eclipse procedure without requiring the Eclipse application be open [5]. This necessitates the required JAR files of an Eclipse distribution be available. When the command to execute the ANT tasks is run from a command line prompt, the headless build acts as though the user had run the ANT task from an Eclipse user interface.

Once we had these components complete along with the necessary JAR files for ANT tasks, ATL transformations, XSLT transformations, and Papyrus for the SysML meta-model, we can develop the visual part of our GUI. We decided to develop this GUI using php to make it simpler to develop and maintain. Also, being web-based means that updates to the

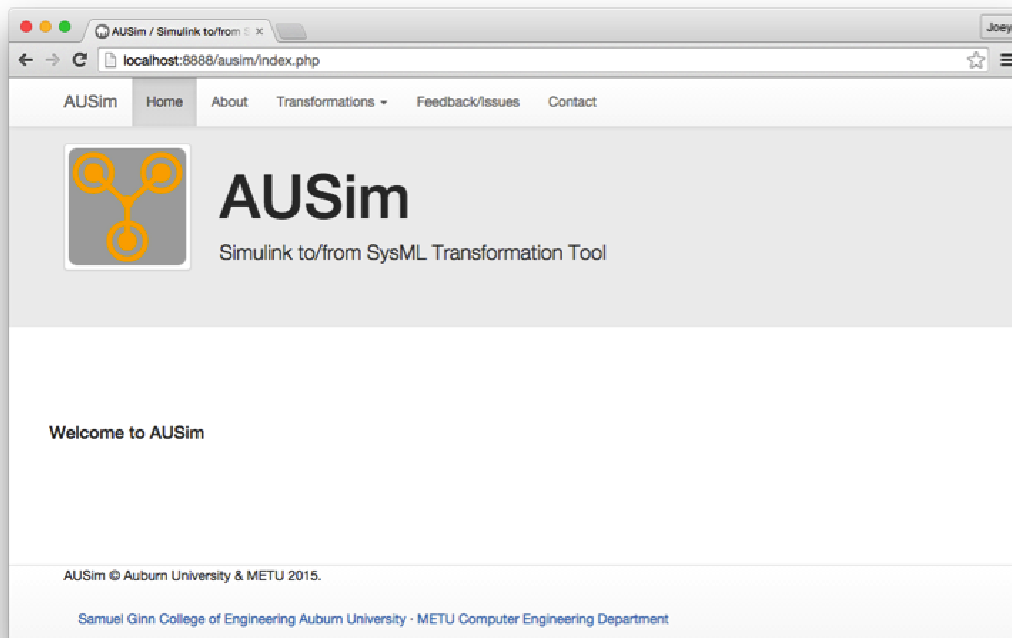

Figure 6.22: User Interface Main Page

application will not require users to install patches or new versions on their local computers. The user experience for this GUI can be seen in Figures 6.22, 6.23, 6.24, 6.25, and 6.26.

From the main page, the user selects which type of transformation is desired, whether from Simulink to SysML or from SysML to Simulink. As more modeling paradigms are added to our transformation process, these options will subsequently be expanded. The user then uploads a Simulink model from their local computer. A progress bar will display while the transformation is executing. Finally, a user will be shown a message if the transformation is successful. If errors are encountered, an error message will display that the model was not successfully transformed. Regardless of whether the transformation is a success or a failure, the user will also be shown the console output to see any important details of the transformation. This console message is the concatenation of all messages the user would have received if they had run the individual transformations in Eclipse.

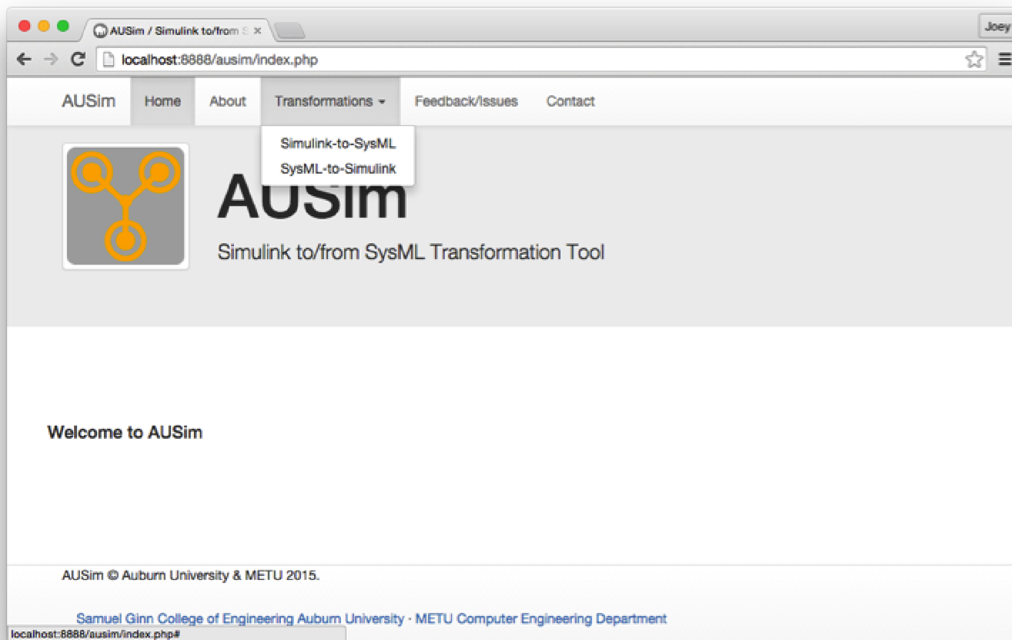

Figure 6.23: Selection of Transformation Type

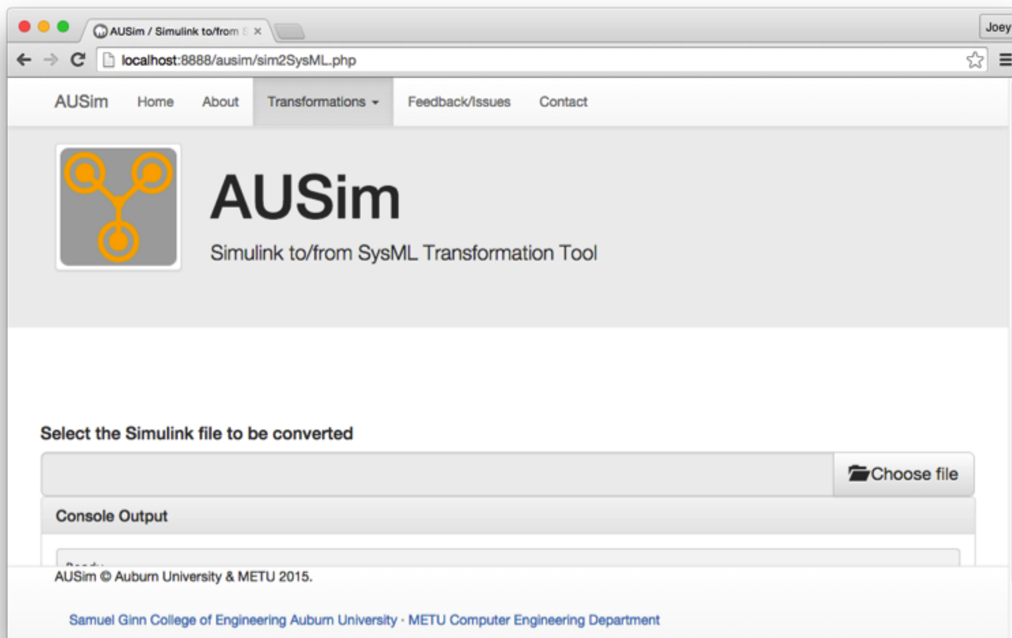

Figure 6.24: Select a Model to Transform

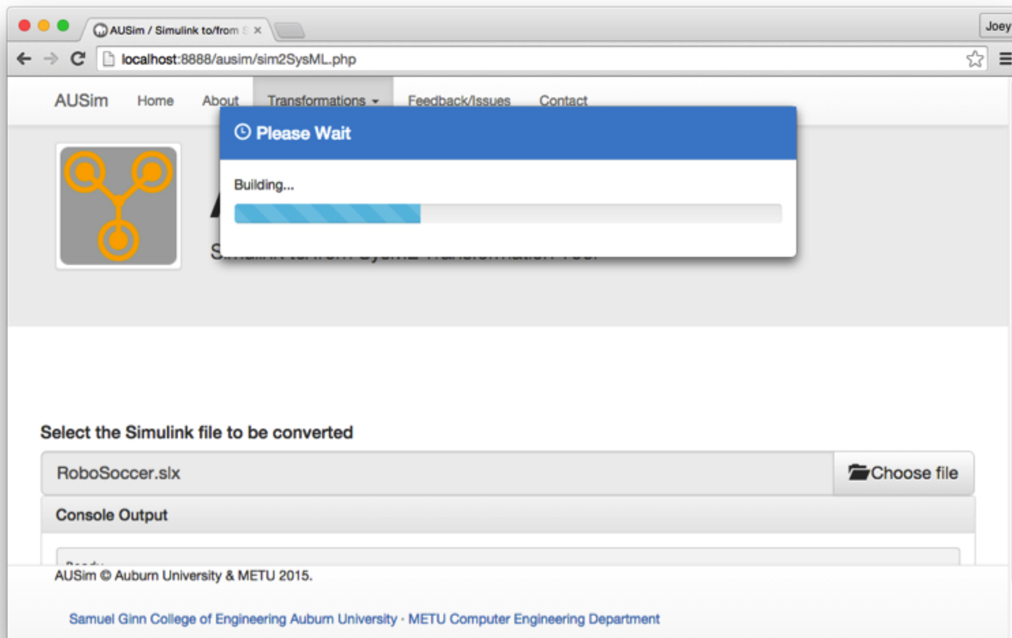

Figure 6.25: Processing the Transformation

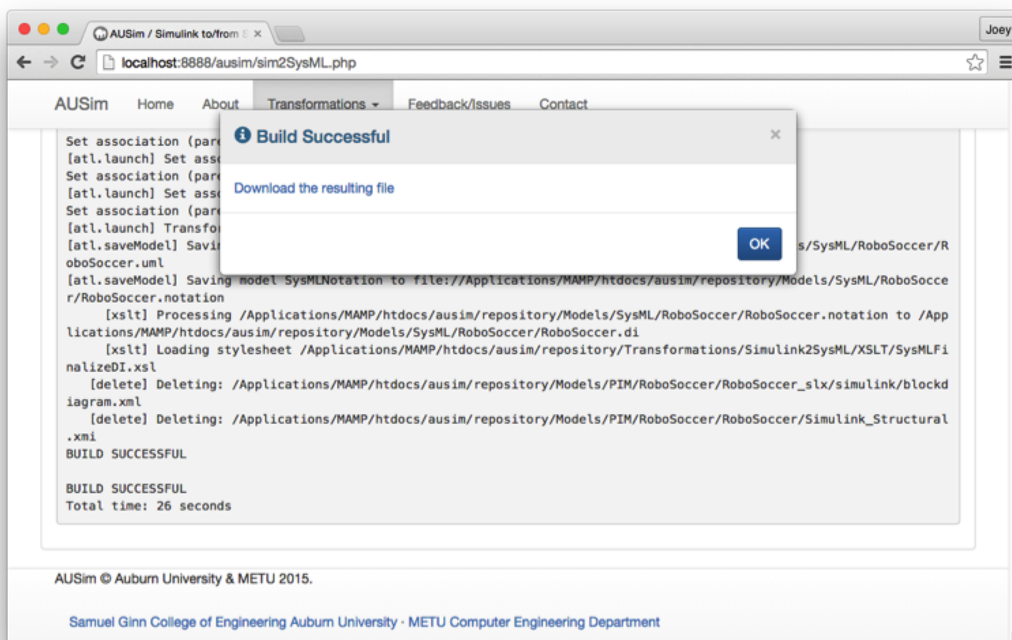

Figure 6.26: Successful Transformation

6.4 Lessons Learned

In this section we detail some of the valuable understanding that was gained in various topics related to computer science and software engineering.

6.4.1 Model Driven Engineering

In creating a process for transforming a model from one domain into another we were able to focus a great deal of time to developing tools and technologies that are primary components of MDE.

Meta-modeling

This process required us to develop two completely new meta-models. Additionally, we were able to research and understand major components of a third, already defined meta-model. This allowed us to gain an understanding of how meta-models describe models and define how they are structured. We were also able to understand the use of meta-meta-models for defining the meta-models in a system.

Model Transformation

The core of this work was to develop a model transformation process. Therefore, it is only reasonable that we would have gained experience in generating a target model from a source model. Specific technologies and tools will be discussed later in this section. In general, we were able to deepen our knowledge of how to define a mapping from a source model Class element to a target model Class. While this often meant looking at particular objects, the desire is to develop a general rule for translating all Objects of a certain type that have specific characteristics into a set of target elements.

6.4.2 Reverse Engineering

We also gained a non-trivial amount of understanding for being able to break down the structure of a given tool or technology to be able to re-create it using other methods. This work required us to be able to understand the inner structure of a Simulink model in order to be able to gain access to the details of the Objects present in the model. Once we had an understanding of the structure of Simulink models, we also needed to perform a similar process to understand the structure of a SysML model in Papyrus. By having knowledge of the structure of a Papyrus model, we were able to generate the necessary Objects to create a SysML model viewable in Eclipse.

6.4.3 Creating a Hybrid Nature for Transformations

Initially, we intended to use ATL as the only transformation technology in the process. Early, we found that we would need a method for taking the values in the XML tags and converting them to attributes of the XML tags for use in the ATL transformations. A simple text parser was thought to be a good option, initially. Early in the development, however, we decided that there was greater potential to use an XML-to-XML translator. In this way, the task of maintaining an appropriate XML structure would be handled by the translator technology, rather than a text parser that we had developed. Once we had discovered the usefulness of creating markup language documents from XML documents using XSLT, it became apparent that we needed to incorporate this second transformation technology into our process. By using a transformation language to create the source models for our ATL transformations, we were able to be more assured that the format of the models will be readable by our ATL transformation rules. By relying on well-documented existing tools such as XSLT and ATL for all facets of the transformation process, we have greater confidence that the models generated will have a well-defined structure.

Originally, the transformation described in Section 5.4.3 for adding Port objects to the Blocks was implemented using XSLT. However, it was found that when executing all the

transformations together, this transformation was a bottleneck in speed due to the need to save the source model for this transformation as well as the target. As such, a method for performing this transformation in ATL was sought and developed.

XSLT

In developing the XSLT portion of the transformation process, we were able to gain valuable insight into the usefulness of this technology. From a valid XML document, XSLT can be used in a wide variety of applications to create documents in any of a number of markup languages. In its normal use, XSLT takes one XML document and generates a second file. It is possible to take a second XML document using the document() function to generate an output file from more than one source XML file. This feature, along with the ability to pass parameters to the XSLT transformation, was used in the SysML to Simulink transformation to combine the structural elements of the model with the Simulink Settings file described in Section 5.2.2 to produce the output blockdiagram.xml file that would be used to create the .slx Simulink model.

ATL

We were also able to learn a great deal from our time developing the ATL rules for our transformation process. Through the use of tutorials available online, we were able to gain some understanding of the basics of ATL rules. Commonly, ATL makes use of matched rules to generate one or more target objects from a source object. However, ATL rules can also incorporate lazy rules to only generate the target objects when a parent source object forces it to. Additionally, lazy rules can be used to generate multiple target objects from a single source. We were able to leverage this feature to be able to generate multiple Property objects from a single source object in the Defaults model as described in Section 5.4.2. Finally, ATL can have called rules. This type of rule does not require any source object to create desired target objects. We also made use of called rules in the final ATL transformation described

in Section 5.4.6 most notably to create some of the required structure in the .notation file for defining the objects necessary for viewing the SysML model. Additionally, we were able to leverage ATL's ability to create a sequence, which is similar to an array, to use its helper syntax to create multiple Port objects from a single integer value. To our knowledge, this was a new development using ATL in this way. We were unable to find any previous instance of a developer performing a similar operation in ATL.

ANT Tasks

In addition to gaining an understanding of the two transformation languages described, we also added experience using ANT Tasks to our repertoire. All of the transformations developed could be easily executed on a specific source model or models by a user with access to an Eclipse distribution. However, to make this tool useful to many researchers and engineers, we needed a method to execute all the transformations in sequence on a given source model. ANT tasks served as a means to do our transformations, as well as the extracting and compressing of files that would be necessary. One useful feature of the ATL ANT tasks is that the output of an ATL transformation does not need to be saved to the computer's hard drive if it is also the source model of a second ATL transformation. However, a model that is the source to an XSLT transformation must exist on the hard drive and the XSLT transformation ANT task saves the resulting model to the hard drive. This fact resulted in a slower transformation process than was necessary when we first developed our ANT task list. As stated above, the transformation to add Port objects to Blocks was originally developed in XSLT. It was determined that this likely was part of the reason for the slow transformation process due to the extra read/write operations to the hard drive.

Chapter 7

Conclusions

Using a hybrid approach of XSLT and ATL to perform an overall model transformation, we were able to develop a model transformation process for taking well-formed Simulink models and generating useful SysML models. We concurrently developed a transformation process for creating executable Simulink models from SysML models. We envision this work will be useful in a number of different ways. Notably, this model transformation process will be very effective as part of a larger experiment management process for improving the reliability of results obtained from scientific experiments by facilitating an increase in the prevalence of model replicability and reproducibility.

In this chapter, we detail some of the areas of research and technology that we anticipate benefiting from this transformation process. Additionally, we describe some of the ongoing developments for this work.

7.1 Benefits

Much potential exists for individuals to add to the wealth of human knowledge and insight using a model transformation process presented here. This potential is not limited to any particular domain. Rather much could be gained in many segments within academic research, the government sector, or private industry. In this section, we discuss some of the potential for our transformation process.

7.1.1 Model Replicability

As discussed in Section 2.2.1, the task of replicating models can be tedious and time-consuming. Also mentioned briefly was that much potential value lies in the ability to

replicate computer models. In this section, we consider some of the possible areas where model replicability will be an invaluable resource.

Validation of Experimental Results

By having a means of transforming a model from a PSM to a PIM, researchers performing experiments with computer models will be able to share the source of their models with a larger scientific community. Other researchers will not require intimate knowledge of the platform used for the original researchers' experiments in order to repeat the experiments. This will greatly assist in the ability for members of the scientific community to verify the accuracy of results obtained using computer models.

Experiment Variability

In addition to being able to verify the results of previous researchers' experiments, other studies can be performed by altering the settings for the independent variables used in the original experiment. Again, by having a PIM representation, knowledge pertaining to the platform used in the original experiment will not be necessary.

Experiment Management

Using a PSM to PIM transformation process, a database of PIMs could be developed. This database can then be accessed by an experiment management utility that will allow scientists to run experiments using any of the models stored in the database. The user interface for this utility could be designed such that the researchers would not require any skills related to model development in order to perform their experiments. The experiments could be run from the user interface on any of the platforms for which we have developed a corresponding PIM to PSM transformation process.

This database can also be used to store information about the experiments that have been run:

- Researcher information
- Model used for experiment
- Platform used for simulation
- Experiment settings
- Results from the simulation

Security of Intellectual Property

Often, model developers are hesitant to allow others to have access to their source code due to the possibility of having their ideas and solutions to certain problems taken or used without permission. A model transformation of a developer's source code in a particular platform into a version that is platform-neutral will help to alleviate some of these concerns. Especially if the developer is allowed to tailor what aspects of the source code are included in the transformation, then it becomes easier to share the functionality of the model without also sharing sensitive information.

7.1.2 Model Driven Engineering

As discussed in Section 2.2.2, MDE involves the development of software systems using various modeling methodologies. Notable among these methodologies are metamodeling and model transformation.

A process for transforming an existing PSM into a PIM, and vice versa can enhance the ability for developers to create software solutions using MDE. Some of the ways these enhancements can be seen are as follows:

- Collaborative development
  - A software development team can be able to collaborate without requiring all team members to have experience with the same modeling platform. A team member

working in one environment could transform the model into a PIM. Other team members could then transform the PIM into a PSM using an environment with which he has more experience. In this way, the developers that have the most insight into the solution can be assigned to a project without needing to spend time learning the specific technology of the modeling platform.

- Software Process
  - The design phase of a software solution could be done using a platform neutral model, then a transformation could be performed to obtain the software solution. In this case, the model designers would not require knowledge of any specific platform.
  - The requirements of a solution can be checked against the final solution by using SysML for the representation. Requirements can be defined using diagrams in SysML. These requirements can then be checked against the solution using SysML standards.

7.2 Future Work

The scope of this work was initially limited to the generation of structural SysML models from Simulink models and creating executable Simulink models when given the structural components of a SysML model. In this section, we attempt to detail some areas for expanding this work to encapsulate a complete transformation process

7.2.1 Simulink MATLAB and SysML ACT Diagrams

One of the useful features of Simulink models is the ability to incorporate MATLAB code embedded in S-Function blocks. Similarly, SysML makes use of Activity Diagrams to describe the operation inside of a SysML block. To expand the existing model transformation process to be able to completely transform all aspects of a source Simulink model into an

equivalent SysML model, we will need to be able to transform the MATLAB code of an S-Function block into a representation using an ACT diagram in SysML.

Figure 7.1 shows the meta-model for the Stateflow section of a Simulink model. This section contains information about state machine behavior and embedded code scripts. Many of these objects will not require transformation into the SysML domain. However, their structure and data will need to be understood for the purposes of transformation into Simulink from a SysML Block and ACT diagram.

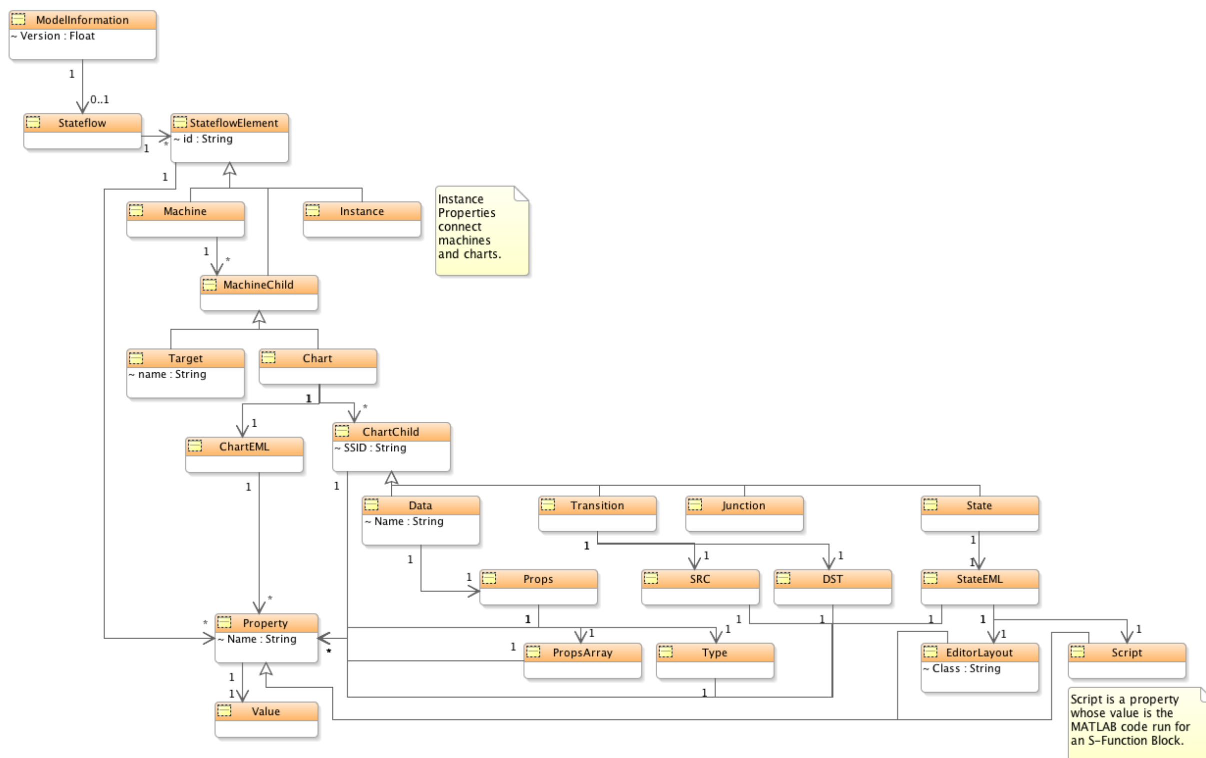

Figure 7.1: Meta-Model of StateFlow Section of SLX Formatted Simulink File

Currently, we have only defined the structure of the Simulink file using our meta-models, including the structure of the Stateflow section. The script that is the source code of an S-Function Block is stored in the Simulink file as a single String value containing MATLAB code. We will need to develop an automated transformation process to parse the code and develop the equivalent ACT Diagram. One potential benefit from addressing the need to

translate Matlab code into SysML Activity Diagrams could be the development of coding standards for Simulink model developers.

We believe strongly that much potential exists to first develop the transformation process from SysML ACT Diagrams into Simulink MATLAB code. By generating a concatenated string of MATLAB code from an object oriented design of an ACT Diagram, we can learn many lessons that will then be useful for developing the "reverse" process from MATLAB code to an ACT Diagram. With this plan, we have developed a potential addition to the Intermediate Source meta-model defined in Section 5.1.2. This addition is shown in Figure 7.2 and the entire Intermediate meta-model can be seen in Figure 7.3. In the potential transformation rules for this aspect of the modified transformation process, we anticipate all nodes from the source SysML model being transformed as Node objects in the Intermediate model. The Next value pointing to the successive Node would be set by evaluating the Connector information from the SysML model. The eventual transformation to Simulink would involve stepping through the Nodes in order and generating a string to create the resulting MATLAB code.

7.2.2 State Machines in Simulink

Another area of potential growth for this transformation process is in the area of Simulink State Machines. We have yet to encounter a state transition example at this point in our research. Our case study does not have any state transitions in its Stateflow section. We plan to seek or create a simple example for the purposes of understanding how the Stateflow section of a Simulink model handles state machines. Once this understanding is complete, the process of adding the necessary elements to the meta-models and transformation rules can be accomplished.

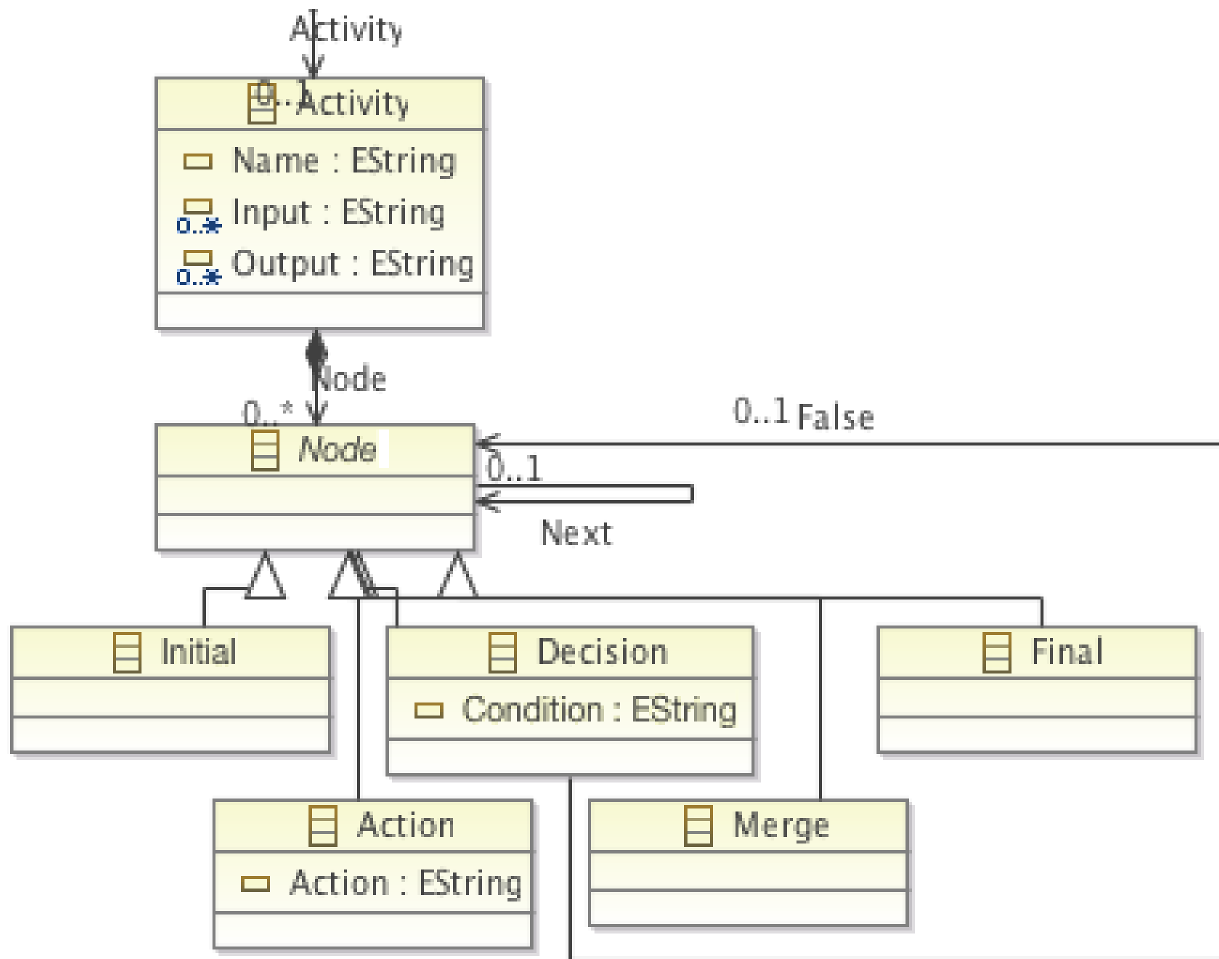

Figure 7.2: Meta-Model of Activity in Intermediate Models

7.2.3 Other Modeling Paradigms

Finally, in order to facilitate a robust system for use by many different researchers with many distinct requirements, we will need to develop the means to transform models from SysML into various different modeling platforms. Work has been accomplished in developing a similar process to generate Repast models from existing SysML models. By being able to generate simulation models in other paradigms, simulation experiments can be tailored to be executed in particular domains. In this way, the results obtained can be verified and validated with more confidence.

Figure 7.3: Meta-Model of Intermediate Models

Bibliography

[1] Ant tasks. https://ant.apache.org/manual/tasksoverview.html. [Online; accessed 23-November-2015].

[2] Ant tasks - unzip. https://ant.apache.org/manual/Tasks/unzip.html. [Online; accessed 23-November-2015].

[3] Ant tasks - xslt. https://ant.apache.org/manual/Tasks/style.html. [Online; accessed 23-November-2015].

[4] Ant tasks - zip. https://ant.apache.org/manual/Tasks/zip.html. [Online; accessed 23-November-2015].

[5] Eclipse headless build. http://www.oracle.com/technetwork/middleware/weblogic-portal/overview/wlp-headless-build-167758.html. [Online; accessed 23-November-2015].

[6] Eclipse modeling framework. http://www.eclipse.org/modeling/emf/. [Online; accessed 23-August-2013].

[7] The moose book. http://www.themoosebook.org/book/internals/fame/subject-model-meta-model/. [Online; accessed 23-August-2013].

[8] Omg systems modeling language. http://www.omgsysml.org/. [Online; accessed 7-November-2013].

[9] Sysml ecore. http://www.eclipse.org/papyrus/0.7.0/SysML.

[10] Uml ecore. http://www.eclipsezone.com/eclipse/forums/t92860.html, April 2007. [Online; accessed 7-October-2013].

[11] Xml. http://www.w3.org/TR/xml/, November 2008. [Online; accessed 17-September-2013].

[12] Taverna. http://www.taverna.org.uk/, 2009. [Online; accessed 5-February-2016].

[13] Ant tasks - atl. http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Tools\#ATL_ant_tasks, 2012. [Online; accessed 23-November-2015].

[14] Repast. http://repast.sourceforge.net/, March 2012. [Online; accessed 12-August-2013].

[15] Sysml. http://www.omg.org/spec/SysML/1.3/, June 2012. [Online; accessed 12-August-2013].

[16] Atl tutorials. http://www.eclipse.org/atl/documentation/basicExamples_Patterns/, 2013. [Online; accessed 7-October-2013].

[17] Atlas transformation language. http://www.eclipse.org/atl/, 2013. [Online; accessed 23-August-2013].

[18] Eclipse - papyrus. http://www.eclipse.org/papyrus/, 2013. [Online; accessed 7-October-2013].

[19] Eclipse modeling framework tools - ecore tools. http://www.eclipse.org/modeling/emft/?project=ecoretools, 2013. [Online; accessed 7-October-2013].

[20] Model driven architecture. http://www.omg.org/mda/specs.htm, 2013. [Online; accessed 17-September-2013].

[21] Modelio open project. http://modelio-open.sourceforge.net/, 2013. [Online; accessed 7-October-2013].

[22] Object management group. http://www.omg.org/, 2013. [Online; accessed 17-September-2013].

[23] Rational rhapsody architect for systems engineers. http://www-03.ibm.com/software/products/us/en/ratirhaparchforsystengi/, 2013. [Online; accessed 7-October-2013].

[24] Save a model. http://www.mathworks.com/help/simulink/ug/saving-a-model.html, 2013. [Online; accessed 27-August-2013].

[25] Simulink. http://www.mathworks.com/products/simulink/, March 2013. [Online; accessed 30-April-2013].

[26] Visual paradigm for uml. http://www.visual-paradigm.com/product/?favor=vpuml, 2013. [Online; accessed 7-October-2013].

[27] Kepler project. https://kepler-project.org/, 2015. [Online; accessed 5-February-2016].

[28] List of simulink blocks. http://www.mathworks.com/help/simulink/blocklist.html, 2015. [Online; accessed 28-October-2015].

[29] Php. https://secure.php.net/, October 2015. [Online; accessed 28-October-2015].

[30] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41, 2003.

[31] D. Balasubramanian, A. Narayanan, C. van Buskirk, and G. Karsai. The graph rewriting and transformation language: Great. *Electronic Communications of the EASST*, 1, 2007.

[32] J. Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[33] J. Bézivin, F. Jouault, and D. Touzet. An introduction to the atlas model management architecture. *Rapport de recherche*, (05.01), 2005.

[34] O. Constant, W. Monin, and S. Graf. A model transformation tool for performance simulation of complex uml models. In *Companion of the 30th international conference on Software engineering*, pages 923–924. ACM, 2008.

[35] S. M. Crook, A. P. Davison, and H. E. Plesser. Learning from the past: approaches for reproducibility in computational neuroscience. In *20 Years of Computational Neuroscience*, pages 73–102. Springer, 2013.

[36] J. Davis. Gme: the generic modeling environment. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 82–83. ACM, 2003.

[37] A. Davison. Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science & Engineering*, 14(4):48–56, 2012.

[38] J. den Haan. `http://www.theenterprisearchitect.eu/archive/2009/01/15/mde---model-driven-engineering----reference-guide`, January 2009. [Online; accessed 23-August-2013].

[39] D. Djuric, D. Gasevic, and V. Devedzic. The tao of modeling spaces. *Journal of Object Technology*, 5(8):125–147, 2006.

[40] D. L. Donoho, A. Maleki, I. U. Rahman, M. Shahram, and V. Stodden. Reproducible research in computational harmonic analysis. *Computing in Science & Engineering*, 11(1):8–18, 2009.

[41] E. Eessaar. Using metamodeling in order to evaluate data models. *AIKED*, 7:181–186, 2007.

[42] S. Fomel and J. F. Claerbout. Guest editors' introduction: Reproducible research. *Computing in Science & Engineering*, 11(1):5–7, 2009.

[43] S. Fomel and G. Hennenfent. Reproducible computational experiments using scons. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, volume 4, pages IV–1257. IEEE, 2007.

[44] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[45] A. Gargantini, E. Riccobene, and P. Scandurra. Integrating formal methods with model-driven engineering. In *Software Engineering Advances, 2009. ICSEA'09. Fourth International Conference on*, pages 86–92. IEEE, 2009.

[46] E. Huang, R. Ramamurthy, and L. F. McGinnis. System and simulation modeling using sysml. In *Proceedings of the 39th conference on Winter simulation: 40 years! The best is yet to come*, pages 796–803. IEEE Press, 2007.

[47] T. Johnson, A. Kerzhner, C. J. Paredis, and R. Burkhart. Integrating models and simulations of continuous dynamics into sysml. *Transactions of the ASME-S-Computing andInfor Science in Engin*, 12(1):011002, 2012.

[48] F. Jouault and I. Kurtev. On the architectural alignment of atl and qvt. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1188–1195. ACM, 2006.

[49] J. Kaiser. If you fail to reproduce another scientists results, this journal wants to know. *Science*, Feb 2016.

[50] E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. *University of Paderborn*, 2007.

[51] J. P. Kleijnen. An overview of the design and analysis of simulation experiments for sensitivity analysis. *European Journal of Operational Research*, 164(2):287–300, 2005.

[52] J. P. Kleijnen, S. M. Sanchez, T. W. Lucas, and T. M. Cioppa. State-of-the-art review: a users guide to the brave new world of designing simulation experiments. *INFORMS Journal on Computing*, 17(3):263–289, 2005.

[53] L. Lengyel, T. Levendovszky, G. Mezei, T. Vajk, and H. Charaf. Practical uses of validated model transformation. In *EUROCON, 2007. The International Conference on" Computer as a Tool"*, pages 2200–2207. IEEE, 2007.

[54] J. P. Mesirov. Accessible reproducible research. *Science*, 327(5964):415–416, 2010.

[55] V. C. Nguyen and X. Qafmolla. Agile development of platform independent model in model driven architecture. In *Information and Computing (ICIC), 2010 Third International Conference on*, volume 2, pages 344–347. IEEE, 2010.

[56] B. Nosek, G. Alter, G. Banks, D. Borsboom, S. Bowman, S. Breckler, S. Buck, C. Chambers, G. Chin, G. Christensen, et al. Promoting an open research culture: author guidelines for journals could help to promote transparency, openness, and reproducibility. *Science (New York, NY)*, 348(6242):1422, 2015.

[57] P. Nowakowski, E. Ciepiela, D. Harężlak, J. Kocot, M. Kasztelnik, T. Bartyński, J. Meizner, G. Dyk, and M. Malawski. The collage authoring environment. *Procedia Computer Science*, 4:608–617, 2011.

[58] OMG. *OMG: Meta Object Facility (MOF) Specification Version 2.4.1*, August 2011.

[59] OMG. *OMG: Unified Modeling Language (UML) Specification Version 2.4.1*, August 2011.

[60] P. RJ. A scientists nightmare: Software problem leads to five retractions. *Science*, 314(1856):2006, 1856.

[61] J. Rothenberg. *The nature of modeling*, volume 3027. Rand, 1989.

[62] S. M. Sanchez and H. Wan. Better than a petaflop: The power of efficient experimental design. In *Winter Simulation Conference*, pages 60–74. Winter Simulation Conference, 2009.

[63] S. M. Sanchez and H. Wan. Work smarter, not harder: a tutorial on designing and conducting simulation experiments. In *Simulation Conference (WSC), Proceedings of the 2012 Winter*, pages 1–15. IEEE, 2012.

[64] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):0025–31, 2006.

[65] E. Soler, J. Trujillo, E. Fernandez-Medina, and M. Piattini. A set of qvt relations to transform pim to psm in the design of secure data warehouses. In *Availability, Reliability and Security, 2007. ARES 2007. The Second International Conference on*, pages 644–654. IEEE, 2007.

[66] V. Stodden. The scientific method in practice: reproducibility in the computational sciences. 2010.

[67] S. J. Taylor, A. Khan, K. L. Morse, A. Tolk, L. Yilmaz, and J. Zander. Grand challenges on the theory of modeling and simulation. In *Proceedings of the 2013 Symposium on the Theory of Modeling and Simulation. SCS, Vista, CA. To appear*, 2013.

[68] D. Turi, P. Missier, C. Goble, D. De Roure, and T. Oinn. Taverna workflows: Syntax and semantics. In *e-Science and Grid Computing, IEEE International Conference on*, pages 441–448. IEEE, 2007.

[69] D. Waltemath, R. Adams, F. T. Bergmann, M. Hucka, F. Kolpakov, A. K. Miller, I. I. Moraru, D. Nickerson, S. Sahle, J. L. Snoep, et al. Reproducible computational biology experiments with sed-ml-the simulation experiment description markup language. *BMC systems biology*, 5(1):198, 2011.

[70] M. Wimmer, M. Strommer, H. Kargl, and G. Kramler. Towards model transformation generation by-example. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 285b–285b. IEEE, 2007.

[71] L. Yilmaz. Reproducibility in m&s research: issues, strategies and implications for model development environments. *Journal of Experimental & Theoretical Artificial Intelligence*, 24(4):457–474, 2012.

[72] L. Yilmaz and T. Ören. Toward replicability-aware modeling and simulation: Changing the conduct of m&s in the information age. In *Ontology, Epistemology, and Teleology for Modeling and Simulation*, pages 207–226. Springer, 2013.

[73] J. Zhao, J. M. Gomez-Perez, K. Belhajjame, G. Klyne, E. Garcia-Cuesta, A. Garrido, K. Hettne, M. Roos, D. De Roure, and C. Goble. Why workflows breakunderstanding and combating decay in taverna workflows. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–9. IEEE, 2012.

Appendices

Appendix A

ATL Examples

Listing A.1: ATL Transformation Rule for Block

```
rule BlockWithSimPort2Block {
        from
                b: Simulink!Block (not b.isSubsystemBlock and b.
                    isPortedBlock)
        to
                dst: Source!Block(
                        Type <- b.BlockType,
                        Name <- b.Name,
                        ID <- b.SID,
                        Property <- b.Property->
                                union(b.Property->select(pos | pos.
                                    isFormattedPositionProperty)->collect(e
                                     | e.Value.Entry))->
                                union(Source!Property.allInstancesFrom('
                                    defaults')->
                                        select(dp | dp.
                                            refImmediateComposite().Type =
                                            b.BlockType
                                                and not b.Property->exists
                                                    (bp | bp.Name = dp.Name
                                                    ))->
                                        collect(blp | thisModule.
                                            defaultProperty(blp))),
                        Port <- b.Port
                )
}
```

Listing A.2: ATL Transformation Helper to Generate Sequence

```
helper def : generateSequence(num : Integer): Sequence(Integer) =
        if num = 0 then
                Sequence{}
        else if num = 1 then
                        Sequence{1}
                else
                        thisModule.generateSequence(num - 1).append(num)
                endif
        endif;
```

Listing A.3: ATL Transformation Rule to Create In Port

```
lazy rule createInPort {
        from
                i : String -- form of {num} + thisModule.nameDelim + {Name
                   } + thisModule.typeDelim + {Type}
        to
                p : Source!Port(
                        Direction <- 'in',
                        Num <- i.substring(1, i.indexOf(thisModule.
                           nameDelim)).toInteger(),
                        Name <- i.substring(i.indexOf(thisModule.nameDelim
                           ) + thisModule.nameDelim.size() + 1, i.indexOf(
                           thisModule.typeDelim)),
                        Type <- i.substring(i.indexOf(thisModule.typeDelim
                           ) + thisModule.typeDelim.size() + 1, i.size())
                )
}
```

Listing A.4: ATL Transformation Rule for Block

```
rule Block2Block {
        from
                bi: Source!Block
        to
                bo: Source!Block (
                        Name <- bi.Name,
                        ID <- bi.ID,
                        Type <- bi.Type,
                        Property <- bi.Property,
                        Port <- thisModule.generateSequence(bi.
                            getInportCount)->collect(i | thisModule.
                            createInPort(i.toString() + thisModule.
                            nameDelim + bi.getPortName(i, 'in') +
                            thisModule.typeDelim + bi.getPortType))->
                                 union(thisModule.generateSequence(bi.
                                    getOutportCount)->collect(o |
                                    thisModule.createOutPort(o.toString() +
                                     thisModule.nameDelim + bi.getPortName(
                                    o, 'out') + thisModule.typeDelim + bi.
                                    getPortType)))
                )
}
```

Listing A.5: ATL Transformation Rules for Lines

```
lazy rule Line2Connection {
        from
                l: Simulink!Line
        to
                c: Source!Connection (
                        Name <- l.Property->select(sl | sl.Name = 'Name')
                           ->
                                collect(slv | slv.Value.Value)->any(sls |
                                   true),
                        src <- thisModule.findPort(l.Property->select(sl |
                            sl.Name.toLower() = 'src')->
                                collect(slv | slv.Value.Value)->any(sls |
                                   true)),
                        dst <- thisModule.findPort(l.Property->select(dl |
                            dl.Name.toLower() = 'dst')->
                                collect(dlv | dlv.Value.Value)->any(dls |
                                   true))
                )
}
```

Listing A.6: ATL Transformation Rules for Branches

```
lazy rule Branch2Connection {
        from
                b: Simulink!Branch
        to
                c: Source!Connection (
                        Name <- b.Property->select(sb | sb.Name = 'Name')
                            ->
                                collect(sbv | sbv.Value.Value)->any(sbs |
                                   true),
                        src <- thisModule.findPort(b.getSrc),
                        dst <- thisModule.findPort(b.Property->select(db |
                            db.Name.toLower() = 'dst')->
                                collect(dbv | dbv.Value.Value)->any(dbs |
                                   true))
                )
}
```

Listing A.7: ATL Transformation Rule for Connections

```
rule System2System {
        from
                bi: Source!System
        to
                bo: Source!System (
                        Name <- bi.Name,
                        ID <- bi.ID,
                        Property <- bi.Property,
                        Block <- bi.Block,
                        Port <- bi.Port,
                        Connection <- Simulink!Line.allInstancesFrom('
                           simLines')->
                                        select(sl | sl.getIDLine = bi.ID
                                           and sl.Branch->size() = 0)->
                                        collect(slp | thisModule.
                                           Line2Connection(slp))->
                                        union(Simulink!Branch.
                                           allInstancesFrom('simLines')->
                                        select(sb | sb.getIDBranch = bi.ID
                                            and sb.Branch->size() = 0)->
                                        collect(sbp | thisModule.
                                           Branch2Connection(sbp)))
                )
}
```

Listing A.8: ATL Transformation Rule for System to Function

```
rule System2FunctionBlock {
        from
                ssys: Source!System (ssys.isFunctionBlock)
        to
                tsys: Source!Block (
                        Name <- ssys.Name.regexReplaceAll('MATLAB', 'Code'
                            ),
                        ID <- ssys.ID,
                        Property <- ssys.Property,
                        Type <- 'Function',
                        Port <- ssys.Port
                )
}
```

Listing A.9: ATL Helper for System to Function

```
helper context Source!System def: isFunctionBlock: Boolean =
        if self.hasBlockOfType('S-Function') and self.hasBlockOfType('
           Demux')
                and self.hasBlockOfType('Terminator') then
                self.countSFunctionPorts('in') = self.countPortBlocks('In'
                   )
                and self.countSFunctionPorts('out') = self.countPortBlocks
                   ('Out') + 1
                and self.connectionExists('S-Function', 'out', 1, 'Demux',
                    'in', 1)
                and self.connectionExists('Demux', 'out', 1, 'Terminator',
                    'in', 1)
                and self.countSFunctionPorts('in') = self.
                   countInConnections
                and self.countPortBlocks('Out') = self.countOutConnections
        else
                false
        endif;

helper context Source!System def: hasBlockOfType (BlockType : String):
   Boolean =
        self.Block->select(b | b.oclIsTypeOf(Source!Block))->exists(d | d.
           Type = BlockType);
```

Listing A.10: ATL Helper for System to Function (Cont'd)

```
helper context Source!System def: countSFunctionPorts (dir : String):
   Integer =
        self.Block->select(b | b.oclIsTypeOf(Source!Block))->
                select(sf | sf.Type = 'S-Function')->
                collect(bp | bp.Port)->flatten()->select(p | p.Direction =
                    dir)->size();

helper context Source!System def: countPortBlocks (dir : String): Integer
   =
        self.Block->select(b | b.oclIsTypeOf(Source!Block))->select(p | p.
           Type = dir + 'port')->size();
```

Listing A.11: ATL Helper for Function Elements

```
helper context Source!Port def: isPartOfFunction: Boolean =
        if self.refImmediateComposite().refImmediateComposite().
           oclIsTypeOf(Source!System) then
                self.refImmediateComposite().refImmediateComposite().
                   isFunctionBlock
        else
                false
        endif;
```

Listing A.12: ATL Rule to Create SysML Block

```
rule Block2Block {
        from
                srcb: Source!Block (not srcb.isPortBlock and not srcb.
                   isFunction and not srcb.hasBlockType)
        to
                sysb: SysML!Class (
                        name <- srcb.Name.regexReplaceAll('\n', '␣'),
                        ownedAttribute <- srcb.Port
                                ->append(thisModule.hChildAssoc(srcb))
                                ->union(srcb.Property)
                ),
                bb : SysML!Block (
                        base_Class<-sysb
                )
        ...
```

Listing A.13: ATL Rule to Create SysML Block (Cont'd)

```
do {
        sysb.name.println();
        thisModule.newBlock(srcb);
        if (not thisModule.blockList.includes(srcb.refImmediateComposite()
           .ID)) {
                thisModule.blockList<-thisModule.blockList.append(srcb.
                   refImmediateComposite().ID);
                thisModule.blockPropertyCount<-thisModule.
                   blockPropertyCount.append(thisModule.blockPropInitCount
                   );
                thisModule.blockOperationCount<-thisModule.
                   blockOperationCount.append(thisModule.blockOpInitCount)
                   ;
                thisModule.blockHeight<-thisModule.blockHeight.append(
                   thisModule.blockInitHeight);
                thisModule.inPortBDDListNextY<-thisModule.
                   inPortBDDListNextY.append(thisModule.inPortInitY);
                thisModule.outPortBDDListNextY<-thisModule.
                   outPortBDDListNextY.append(thisModule.outPortInitY);
                thisModule.inPortIBDParentListNextY<-thisModule.
                   inPortIBDParentListNextY.append(thisModule.inPortInitY)
                   ;
                thisModule.outPortIBDParentListNextY<-thisModule.
                   outPortIBDParentListNextY.append(thisModule.
                   outPortInitY);
                thisModule.inPortIBDChildListNextY<-thisModule.
                   inPortIBDChildListNextY.append(thisModule.inPortInitY);
                thisModule.outPortIBDChildListNextY<-thisModule.
                   outPortIBDChildListNextY.append(thisModule.outPortInitY
                   );
        }...
```

Listing A.14: ATL Rule to Create SysML Block (Cont'd)

```
    if (not thisModule.blockList.includes(srcb.ID)) {
            thisModule.blockList<-thisModule.blockList.append(srcb.ID)
               ;
            thisModule.blockPropertyCount<-thisModule.
               blockPropertyCount.append(thisModule.blockPropInitCount
               );
            thisModule.blockOperationCount<-thisModule.
               blockOperationCount.append(thisModule.blockOpInitCount)
               ;
            thisModule.blockHeight<-thisModule.blockHeight.append(
               thisModule.blockInitHeight);
            thisModule.inPortBDDListNextY<-thisModule.
               inPortBDDListNextY.append(thisModule.inPortInitY);
            thisModule.outPortBDDListNextY<-thisModule.
               outPortBDDListNextY.append(thisModule.outPortInitY);
            thisModule.inPortIBDParentListNextY<-thisModule.
               inPortIBDParentListNextY.append(thisModule.inPortInitY)
               ;
            thisModule.outPortIBDParentListNextY<-thisModule.
               outPortIBDParentListNextY.append(thisModule.
               outPortInitY);
            thisModule.inPortIBDChildListNextY<-thisModule.
               inPortIBDChildListNextY.append(thisModule.inPortInitY);
            thisModule.outPortIBDChildListNextY<-thisModule.
               outPortIBDChildListNextY.append(thisModule.outPortInitY
               );
    }
    thisModule.CreateBDDBlockShape(SysMLnot!Shape.allInstancesFrom('
       notDef')->select(s | s.type='shape_sysml_block_as_classifier')
       ->first(), sysb, false, srcb.ID, 4);
    } }
```

Listing A.15: ATL Rule to Create SysML Block Shape

```
rule CreateBDDBlockShape(src : SysMLnot!Shape, ele : SysML!Class, hasOp:
   Boolean, bid : String, row : Integer) {
to
trg: SysMLnot!Shape (
        type<-thisModule.delimChar+'7016'+thisModule.delimChar+src.type,
        fontName<-src.fontName,
        fontHeight<-src.fontHeight,
        lineColor<-src.lineColor,
        eAnnotations<-src.eAnnotations->collect(e | thisModule.
           ShapeeAnnotations(e)),
        children<-src.children->
                select(d | d.oclIsTypeOf(SysMLnot!DecorationNode) and d.
                   layoutConstraint.oclIsUndefined())->collect(n |
                   thisModule.ShapeDecorationNode(n))->
                union(src.children->select(d | d.oclIsTypeOf(SysMLnot!
                   DecorationNode) and not d.layoutConstraint.
                   oclIsUndefined())->collect(n | thisModule.
                   ShapeDecorationNodeHasLC(n)))->
                union(src.children->select(l | l.oclIsTypeOf(SysMLnot!
                   ListCompartment))->
                      reject(p | p.type='
                         compartment_sysml_property_as_list')->
                      reject(o | not hasOp and o.type='
                         compartment_uml_operation_as_list')->
                      collect(c | thisModule.ShapeListCompartment(c)))->
                union(SysMLnot!Shape.allInstancesFrom('nota')->select(s |
                   s.type.endsWith(thisModule.delimChar + thisModule.BDDID
                    + thisModule.delimChar + '
                   shape_sysml_flowport_as_affixed'))->select(s2 | s2.
                   isChildShapeOfParent(ele)))->prepend(lcProp),
        element<-ele, layoutConstraint<-bnds ), ...
```

Listing A.16: ATL Rule to Create SysML Block Shape

```
...
lcProp: SysMLnot!ListCompartment (
        type<-'compartment_sysml_property_as_list',
        showTitle<-true,
        children<-SysMLnot!Node.allInstancesFrom('nota')->select(s | s.
           type='shape_uml_property_as_label' and s.isChildNodeOfParent(
           ele))
),
bnds: SysMLnot!Bounds(
        x<-thisModule.blockNextX.at(row),
        y<-thisModule.blockNextY.at(row),
        width<-src.layoutConstraint.width,
        height<-src.layoutConstraint.height
)
do {
        thisModule.blockNextX <- thisModule.setBlockX(row, bnds.width);
        thisModule.packageHeight <- thisModule.packageHeight.max(bnds.y +
           bnds.height + 40);
        thisModule.packageWidth <- thisModule.packageWidth.max(bnds.x +
           bnds.width + 40 + (ele.longestLabel * 9));
}
}
```

Appendix B

XSLT Examples

Listing B.1: XSLT Transformation Rules for Simulink Settings

```
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/
   Transform" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.
   org/2001/XMLSchema-instance" xmi:version="2.0" xsi:schemaLocation="
   http://www.omg.org/XMI" xmlns:xalan="http://xml.apache.org/xalan">
  <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"
     xalan:indent-amount="2"/>

  <xsl:variable name="MatchedElementList" select="'|Model|ModelInformation
     |␣...␣|'"/>
  <xsl:variable name="SkippedPropertyList" select="'|NumRootInports|
     NumRootOutports|␣...␣|'"/>

  <xsl:template match="*">
    <xsl:if test="contains($MatchedElementList,␣concat('|',␣local-name(),␣
       '|'))">
      <xsl:copy><xsl:for-each select="@*">
          <xsl:attribute name="{name()}"><xsl:value-of select="."/></
             xsl:attribute>
        </xsl:for-each><xsl:apply-templates/>
      </xsl:copy></xsl:if>
  </xsl:template>

  <xsl:template match="P">
    <xsl:if test="not(contains($SkippedPropertyList,␣concat('|',␣@Name,␣
       '|')))">
      <xsl:copy><xsl:for-each select="@*">
          <xsl:attribute name="{name()}"><xsl:value-of select="."/></
             xsl:attribute>
        </xsl:for-each><xsl:apply-templates/>
      </xsl:copy></xsl:if>
  </xsl:template>
```

Listing B.2: XSLT Transformation for Arrays

```
<xsl:template name="ProcessArrayPortion">
  <xsl:param name="ArrayPortion" select="."/>
  <xsl:param name="delimList" select="."/>
  <xsl:param name="index" select="."/>
  <xsl:choose>
    <xsl:when test="string-length($delimList)␣>␣1">
      <xsl:call-template name="ArraySplit">
        <xsl:with-param name="ArrayList" select="$ArrayPortion"/>
        <xsl:with-param name="delimList" select="substring($delimList,␣
           2)"/>
        <xsl:with-param name="index" select="$index"/>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:if test="string-length($ArrayPortion)␣>␣0">
        <xsl:text>&#xa;</xsl:text>
        <xsl:element name="Entry">
          <xsl:attribute name="Index">
            <xsl:value-of select="$index"/>
          </xsl:attribute>
            <xsl:call-template name="ProcessPrimitiveType">
              <xsl:with-param name="PrimitiveValue" select="$
                  ArrayPortion"/>
            </xsl:call-template>
        </xsl:element>
      </xsl:if>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Listing B.3: XSLT Transformation for Arrays (Cont'd)

```
<xsl:template name="ArraySplit">
  <xsl:param name="ArrayList" select="."/>
  <xsl:param name="delimList" select="."/>
  <xsl:param name="index" select="."/>
  <xsl:choose>
    <xsl:when test="contains($ArrayList,␣substring($delimList,␣1,␣1))">
      <xsl:variable name="ArrayPortion" select="normalize-space(
         substring-before($ArrayList,␣substring($delimList,␣1,␣1)))"/>
      <xsl:call-template name="ProcessArrayPortion">
        <xsl:with-param name="ArrayPortion" select="$ArrayPortion"/>
        <xsl:with-param name="delimList" select="$delimList"/>
        <xsl:with-param name="index" select="$index"/>
      </xsl:call-template>
      <xsl:call-template name="ArraySplit">
        <xsl:with-param name="ArrayList" select="normalize-space(
           substring-after($ArrayList,␣substring($delimList,␣1,␣1)))"/>
        <xsl:with-param name="delimList" select="$delimList"/>
        <xsl:with-param name="index" select="$index+1"/>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:call-template name="ProcessArrayPortion">
        <xsl:with-param name="ArrayPortion" select="$ArrayList"/>
        <xsl:with-param name="delimList" select="$delimList"/>
        <xsl:with-param name="index" select="$index"/>
      </xsl:call-template>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Listing B.4: XSLT Transformation for Properties

```
<xsl:template match="P">
  <xsl:variable name="name" select="@Name"/>
      <xsl:element name="Property">
        <xsl:for-each select="@*">
          <xsl:attribute name="{name()}">
              <xsl:value-of select="."/>
          </xsl:attribute>
        </xsl:for-each>
        <xsl:variable name="value" select="." />
        <xsl:call-template name="ProcessPrimitiveType">
          <xsl:with-param name="PrimitiveValue" select="$value"/>
        </xsl:call-template>
      </xsl:element>
</xsl:template>
```

Listing B.5: XSLT Transformation for DI File

```
<xsl:template name="GetDiagrams" match="/xmi:XMI">
  <xsl:variable name="diags">
    <xsl:for-each select="*">
      <xsl:variable name="tag" select="local-name()"/>
      <xsl:if test="$tag='Diagram'␣and␣@type='BlockDefinition'">
        <xsl:value-of select="@xmi:id"/><xsl:value-of select="$delim"/><
           /xsl:if>
    </xsl:for-each>
    <xsl:for-each select="*">
      <xsl:variable name="tag" select="local-name()"/>
      <xsl:if test="$tag='Diagram'␣and␣@type='InternalBlock'">
        <xsl:value-of select="@xmi:id"/><xsl:value-of select="$delim"/><
           /xsl:if>
    </xsl:for-each>
    <xsl:for-each select="*">
      <xsl:variable name="tag" select="local-name()"/>
      <xsl:if test="$tag='Diagram'␣and␣@type='PapyrusUMLActivityDiagram'
         ">
        <xsl:value-of select="@xmi:id"/><xsl:value-of select="$delim"/><
           /xsl:if>
    </xsl:for-each>
  </xsl:variable>
```

Appendix C

XML Examples

Listing C.1: Simulink XML of Block Structure for Team 1 Strategy

```
<System>
  <Block BlockType="Inport" Name="ball_position" SID="4::1">...</Block>
  <Block BlockType="Inport" Name="power" SID="4::21">
    <P Name="Port">2</P>
  </Block>
  <Block BlockType="Inport" Name="robot_position" SID="4::19">
    <P Name="Port">3</P>
  </Block>
  <Block BlockType="Demux" Name="␣Demux␣" SID="4::23">
    <P Name="Ports">[1, 1]</P>
  </Block>
  <Block BlockType="S-Function" Name="␣SFunction␣" SID="4::22">
    <P Name="Tag">Stateflow S-Function Team_1_Strategy 7</P>
    <P Name="Ports">[3, 3]</P>
    <Port>
      <P Name="PortNumber">2</P>
      <P Name="Name">actuator_force_1</P>
    </Port>
    <Port>
      <P Name="PortNumber">3</P>
      <P Name="Name">actuator_force_2</P>
    </Port>
  </Block>
  <Block BlockType="Terminator" Name="␣Terminator␣" SID="4::25">...</Block>
  <Block BlockType="Outport" Name="actuator_force_1" SID="4::5">...</Block>
  <Block BlockType="Outport" Name="actuator_force_2" SID="4::20">
    <P Name="Port">2</P>
  </Block>
```

Listing C.2: Simulink XML of Line Connections for Team 1 Strategy

```xml
  <Line>
    <P Name="Src">4::1#out:1</P>
    <P Name="Dst">4::22#in:1</P>
  </Line>
  <Line>
    <P Name="Src">4::21#out:1</P>
    <P Name="Dst">4::22#in:2</P>
  </Line>
  <Line>
    <P Name="Src">4::19#out:1</P>
    <P Name="Dst">4::22#in:3</P>
  </Line>
  <Line>
    <P Name="Name">actuator_force_1</P>
    <P Name="Src">4::22#out:2</P>
    <P Name="Dst">4::5#in:1</P>
  </Line>
  <Line>
    <P Name="Name">actuator_force_2</P>
    <P Name="Src">4::22#out:3</P>
    <P Name="Dst">4::20#in:1</P>
  </Line>
  <Line>
    <P Name="Src">4::23#out:1</P>
    <P Name="Dst">4::25#in:1</P>
  </Line>
  <Line>
    <P Name="Src">4::22#out:1</P>
    <P Name="Dst">4::23#in:1</P>
  </Line>
</System>
```

Listing C.3: Modified Simulink XMI of Block Structure for Team 1 Strategy

```
<System>
  <Block BlockType="Inport" Name="ball_position" SID="4::1">...</Block>
  <Block BlockType="Inport" Name="power" SID="4::21">
    <Property Name="Port" Value="2" />
  </Block>
  <Block BlockType="Inport" Name="robot_position" SID="4::19">
    <Property Name="Port" Value="3" />
  </Block>
  <Block BlockType="Demux" Name="␣Demux␣" SID="4::23">
    <Property Name="Ports" Value="[1,␣1]" />
  </Block>
  <Block BlockType="S-Function" Name="␣SFunction␣" SID="4::22">
    <Property Name="Tag" Value="Stateflow␣S-Function␣Team_1_Strategy␣7" />
    <Property Name="Ports" Value="[3,␣3]" />
    <Port>
      <Property Name="PortNumber" Value="2" />
      <Property Name="Name" Value="actuator_force_1" />
    </Port>
    <Port>
      <Property Name="PortNumber" Value="3" />
      <Property Name="Name" Value="actuator_force_2" />
    </Port>
  </Block>
  <Block BlockType="Terminator" Name="␣Terminator␣" SID="4::25">...</Block>
  <Block BlockType="Outport" Name="actuator_force_1" SID="4::5">...</Block>
  <Block BlockType="Outport" Name="actuator_force_2" SID="4::20">
    <Property Name="Port" Value="2" />
  </Block>
```

Listing C.4: Modified Simulink XMI of Line Connections for Team 1 Strategy

```
  <Line>
    <Property Name="Src" Value="4::1#out:1" />
    <Property Name="Dst" Value="4::22#in:1" />
  </Line>
  <Line>
    <Property Name="Src" Value="4::21#out:1" />
    <Property Name="Dst" Value="4::22#in:2" />
  </Line>
  <Line>
    <Property Name="Src" Value="4::19#out:1" />
    <Property Name="Dst" Value="4::22#in:3" />
  </Line>
  <Line>
    <Property Name="Name" Value="actuator_force_1" />
    <Property Name="Src" Value="4::22#out:2" />
    <Property Name="Dst" Value="4::5#in:1" />
  </Line>
  <Line>
    <Property Name="Name" Value="actuator_force_2" />
    <Property Name="Src" Value="4::22#out:3" />
    <Property Name="Dst" Value="4::20#in:1" />
  </Line>
  <Line>
    <Property Name="Src" Value="4::23#out:1" />
    <Property Name="Dst" Value="4::25#in:1" />
  </Line>
  <Line>
    <Property Name="Src" Value="4::22#out:1" />
    <Property Name="Dst" Value="4::23#in:1" />
  </Line>
</System>
```

Listing C.5: DI File Contents for Team 1 Strategy

```
<di:SashWindowsMngr xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:di="http://
    www.eclipse.org/papyrus/0.7.0/sashdi">
  <pageList>
    <availablePage>
    <emfPageIdentifier href="Team_1_Strategy.notation#_Pp-
        ekPRAEeSBKOq6V7iV0g"/></availablePage>
    <availablePage>
    <emfPageIdentifier href="Team_1_Strategy.notation#
        _Pp60MPRAEeSBKOq6V7iV0g"/></availablePage>
    <availablePage>
    <emfPageIdentifier href="Team_1_Strategy.notation#
        _Pp5mEPRAEeSBKOq6V7iV0g"/></availablePage>
  </pageList>
  <sashModel currentSelection="//@sashModel/@windows.0/@children.0">
    <windows>
      <children xsi:type="di:TabFolder">
        <children>
        <emfPageIdentifier href="Team_1_Strategy.notation#_Pp-
            ekPRAEeSBKOq6V7iV0g"/></children>
        <children>
        <emfPageIdentifier href="Team_1_Strategy.notation#
            _Pp60MPRAEeSBKOq6V7iV0g"/></children>
        <children>
        <emfPageIdentifier href="Team_1_Strategy.notation#
            _Pp5mEPRAEeSBKOq6V7iV0g"/></children>
      </children>
    </windows>
  </sashModel>
</di:SashWindowsMngr>
```

Appendix D

ANT Task Example

Listing D.1: ANT Tasks used in Simulink to SysML Transformation

```
<xslt style="${transPath}/XSLT/BlockDiagram2SimulinkStructural.xsl"
        in="${modelPath}/PIM/${rootModelName}/${modelName}_slx/simulink/
           blockdiagram.xml"
        out="${modelPath}/PIM/${rootModelName}/${modelName}/
           Simulink_Structural.xmi"
        force="true">
        <param name="ModelName" expression="${modelName}"/>
</xslt>

<atl.loadModel name="Source" metamodel="MOF" path="${metaModelPath}/
   SimulinkSourceMM.ecore"/>
<atl.loadModel name="Simulink" metamodel="MOF" path="${metaModelPath}/
   SimulinkMM.ecore"/>
<atl.loadModel name="SysML" metamodel="MOF" nsUri="http://www.eclipse.org/
   papyrus/0.7.0/SysML"/>

<atl.loadModel name="simModel" metamodel="Simulink" path="${modelPath}/PIM
   /${rootModelName}/${modelName}/Simulink_Structural.xmi"/>

<atl.launch path="${transPath}/ATL/Source2SourceWConnections.asm">
        <inmodel name="src" model="srcModelWPorts"/>
        <inmodel name="simLines" model="simModel"/>
        <outmodel name="srcwcon" model="srcModelWConnections" metamodel="
           Source"/>
</atl.launch>

<atl.saveModel derived="false" model="SysMLModel" path="${modelPath}/SysML
   /${rootModelName}/${modelName}.uml"/>
<atl.saveModel derived="false" model="SysMLNotation" path="${modelPath}/
   SysML/${rootModelName}/${modelName}.notation"/>
```