MODEL TRANSFORMATION BY DOMAIN-SPECIFIC MODELS

by

Maribel Hudson

A Thesis Submitted to the Faculty of the Electrical and Computer Engineering
In Partial Fulfillment of the Requirements For the Degree of Master of Science
In the Graduate College
THE UNIVERSITY OF ARIZONA

November 18, 2011
STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

Signed: ____________________________
Notes
Section for keeping notes on sources so not everything needs to be remembered.
# Table of Contents

**List of Figures** .......................................................... v

**List of Tables** ............................................................ vii

**Abstract** ..................................................................... 1

**Chapter 1. Introduction** .................................................... 2

  1.1. The Role of Model Transformations .................................. 4
  1.2. Semantically and Structurally Rich Models ...................... 6
  1.3. Contribution of This Thesis ......................................... 8

**Chapter 2. Background** ..................................................... 9

  2.1. Model-based Design .................................................. 9
  2.2. Domain-Specific Modeling and Domain-Specific Modeling Environments ............................................. 10
  2.3. Graph and Model Transformation .................................. 14
     2.3.1. Model Transformation by Example .......................... 15
     2.3.2. Model Transformation by Demonstration ................. 17
  2.4. Graph Rewriting and Transformation (GReAT) ............... 20
     2.4.1. Configuring Transformations in GReAT ................... 21
     2.4.2. Elements of A Rule in GReAT ............................... 22
     2.4.3. GReAT Actions ................................................. 25
     2.4.4. Rule Alternatives in GReAT ................................. 25

**Chapter 3. Methods** ....................................................... 29

  3.1. Problem ............................................................... 29
  3.2. Approach .............................................................. 36
     3.2.1. Artifacts ......................................................... 36
     3.2.2. Generation of \( g \) .............................................. 37
     3.2.3. Self-validation of \( g \) ......................................... 39

**Chapter 4. Results** ....................................................... 46

  4.1. Case Study ............................................................ 46
     4.1.1. Source model \( m_0 \) .......................................... 46
     4.1.2. The generated transformation: \( g \) ......................... 47
     4.1.3. Reuse of \( r_1 \) ................................................. 52

**Chapter 5. Conclusions** .................................................. 59

  5.1. Limitations ............................................................ 60
TABLE OF CONTENTS—Continued

Chapter 6. Future Work ........................................................................... 61

References ............................................................................................... 62
LIST OF FIGURES

FIGURE 2.1. An example class diagram, relating Laptop and ExternalKeyboard classes. ............................................. 9
FIGURE 2.2. An example object diagram, relating instances of the Laptop and ExternalKeyboard classes. ...................... 10
FIGURE 2.3. The necessary steps for performing model transformation by demonstration seen in [51]. .......................... 18
FIGURE 2.4. Example configuration model for a transformation in GReAT from [1]. ....................................................... 22
FIGURE 2.5. An Example of multiple rules making up a transformation in GReAT from [1]. ............................................. 22
FIGURE 2.6. An example rule in GReAT, from [7]. ......................... 23
FIGURE 2.7. Different types of elements in a transformation sequence ................................................................. 26
FIGURE 2.8. Example Block Elements .................................................. 27
FIGURE 3.1. Simple metamodel, demonstrating the declarative nature of semantically and structurally rich models. Stereotypes in the language come from types in GME [31]. ................................. 30
FIGURE 3.2. Instance model of the metamodel shown in Figure 3.1. The internal contents of the Product and Integrator blocks are shown. Each has a collection of Attribute models (squares) and contains other Block models (rectangles). .................................................. 31
FIGURE 3.3. A element from Figure 3.1 with internal elements shown. ................................................................. 32
FIGURE 3.4. A rule to instantiate a structurally correct Constant in Figure 3.1 modeled using the GReAT model transformation language. Stereotypes in the diagram for Folder, Model, etc., are according to the syntax of GReAT .................................................. 34
FIGURE 3.5. Code needed for two different AttributeMappings ................................................................. 35
FIGURE 3.6. This sequence of transformation blocks defines $h$, and when executed on some model $m_0 \models L$, will generate $g \models L_G$ (i.e., $g$ is a GReAT model, generated by a GReAT model). .................................................. 38
FIGURE 3.7. Comparison of original model versus the produced model with null input .................................................. 41
FIGURE 3.8. Constant comparison using XML excerpts of constant element .................................................. 43
FIGURE 3.9. Constant comparison using XML excerpts of port element .................................................. 44
FIGURE 3.10. Constant comparison using XML excerpt of atom element .................................................. 45
Figure 4.1. The source artifact for the case study. 46

Figure 4.2. The transformation $g$, the result of $h(m_0)$ where $m_0$ is found in Figure 4.1. Note that the $h$ creates all objects inside the container in one rule. The screenshots are taken from the generated GReAT transformation. 47

Figure 4.3. The Internal elements and structure of the GeneratedRule from Figure 4.2. 48

Figure 4.4. Model comparison of original and generated models. 49

Figure 4.5. part browser comparison of original and generated models. 50

Figure 4.6. xml comparison of original and generated models. 51

Figure 4.7. End result of GeneratedRule after easy changes are made to reuse. 54

Figure 4.8. Contents of the Rule which will create the displays. 55

Figure 4.9. The final transformation with both rules connected. 55

Figure 4.10. The transformation results of generated transformation. 56

Figure 4.11. The input model for the second test of the transformation found in Figure 4.9. 57

Figure 4.12. The output of the transformation found in Figure 4.9 when Figure 4.11 is given as an input. 58
LIST OF TABLES
When transforming models between various domain-specific modeling languages and tools, simple (domain-specific) concepts in the source or target language may have complex representations as patterns of a model transformation, due to parameters or structure that must exist for valid behavior. For domains such as simulation, control, and embedded systems, these so-called “semantically rich” structures are common. This thesis presents a method to synthesize transformation patterns directly from semantically rich domain-specific models. The goal is not transformations by example; rather, the aim is reusable templates for model transformations composed of complex—but domain-specific—patterns. A case study using a signal processing modeling language is presented, where the insertion of intermediate (complex) filters improves model performance. An additional benefit is that the generated transformation patterns can be shown to be self-validating.
Chapter 1

INTRODUCTION

Model-based design is rapidly becoming the standard for software and systems development. The Unified Modeling Language (UML) [10] permitted software designers to more accurately express system ideas during design phases. Particularly, use cases and sequence models permit an interaction between system designers and the system customer, in order to fully vet requirement prior to investing in design and implementation. Class diagrams permit a language-independent way to describe the static description of a system’s objects, and object diagrams are a way to represent instantiated objects, and their relations at runtime.

Although UML certainly catalyzed the software industry to formalize design artifacts, it has been the emergence of the Model-Driven Architecture (MDA) and UML2 (standardized by the Object Management Group [54]) that has permitted system modeling to grow from the static descriptions seen in object diagrams and class diagrams, to dynamic descriptions of how an object diagram should change. Previous descriptions of dynamics in UML were done using state models [14]. However, the state model approach permits an execution of a particular object (as it responds to messages and events), but does not prescribe a behavior for such events.

The behavior of a system, then, has quite simply still been relegated to a software implementation, and this still largely the case. For system design where the behavior of individual system parts is well known, the approach of domain-specific modeling permits common behaviors to be abstracted into domain-specific types, and the system behavior can be inferred from the structure of the system. This inference is usually carried out through some kind of code generation or software synthesis, and examples of this approach are numerous. For instance, the domain of digital
logic design can be abstracted into several kinds of gates (e.g., **AND**, **XOR**), and the domain-specific model is the composition of these various gates into a system.

The advantages of a domain-specific approach to model-based design are many. For instance, using domain-specific types means that boilerplate work (in software, usually accounted for by cloning) is not needed, as common elements are abstracted into the system types. This reduces errors in cloning, as well as simplifies system specification. Another benefit is that the system specification can usually be done more rapidly, since the domain-specific model, as constructed, will consist of models that are directly mapped to a particular application domain. An important benefit is that the domain-specific language is normally formally defined, which means that the design of such a language can prevent the modeler from creating models that are not well-formed.

Clearly, if the trend in practice is for more system design to be performed using models, and for the software that executes the system to be synthesized from those models, then significant effort is required to construct the software synthesizers (called either model compilers, model interpreters, etc.). A common way in which model compilers have been written is to treat the compiler just like that of a textual language, and use common textual languages (C#, Java, C++, etc.) to parse the model tree and synthesize the output code.

However, a recent trend has been to employ the theory of graph transformations to transforming models to structures that can be more easily compiled. This technology, called **model transformations**, is a rich field of investigation, and several tools and approaches already exist, which can examine patterns in a built model, and transform those patterns into another modeling language.
1.1 The Role of Model Transformations

Model transformations can play a key role in the well-studied domains discussed earlier. For instance, model transformations can refactor the structure of a computation in combinational logic circuits [37], can enhance the safety of a control system by automatically installing supervisory observers at key decision points [44], and can stabilize the behavior of event-driven component based designs by introducing time-triggered semantics [47].

There exist different types of model transformation which are mentioned in [43]. The first of these is direct model manipulation; for this type of transformation, the user manipulates the elements manually with access to the set of procedural APIs. The second form of model transformation is called intermediate representation. In this model transformation technique the visual model is interpreted into an XML version and then the exported xml is transformed. The final type of transformation is transformation language support. In these transformations the user is given a transformation language whose elements represent mechanisms available for applying a transformation on the graph. This thesis focus on the simplification of transformations which have language support.

For model transformations of language support there is a generally agreed upon structure for the flow of model transformation. Figure 1.1 extends the visual definition given in [17]. The top figure shows the different parts of a model transformation. The source model is the model to be changed by the transformations and the target model is the final output of the transformation. In model driven design models must conform to a metamodel which governs the rules and behaviors of the specific set of models. The source and target models are no exception to this rule and as such they must have metamodels which they conform to. The transformation engine uses the metamodels to define the transformations which should happen and the elements that will be

\footnote{Many of these examples have existing model transformations solutions, while others have been demonstrated with generic algorithms, but not through a model transformation approach.}
Figure 1.1. (a) Principal parts of model transformation shown in [17]; (b) extensions on transformation execution and transformation definition which apply to the work done in this paper.
involved in the transformation. Finally the transformation engine will read a give
source model and execute the transformation definition created to produce the target
model.

Figure 1.1 has two addition sections to illustrate that a transformation engine
is the platform which performs the transformation. The transformation platform
reads the patterns and the transformation and then does the search necessary in the
source model. Examples of transformation engines include GReAT [1][2], FUJABA
seen in [38] and [29], VIATRA2 [16] and PROGRES [42] [56]. Finally the figure
illustrates the make of the transformation definition. Most transformation definitions
are specific languages which require a metamodel and a transformation model. This
is particularly accurate when dealing with visual transformation languages. Specific
information on how the model transformation language can be abstracted to allow
flexibility with metamodels can be found in [35] and [36].

This thesis considers endogenous transformations (where the host and target
graphs are models that conform to the same metamodel), though the presented tech-
niques may be extended for exogenous transformations with some minor modifica-
tions. For endogenous transformations where models are instantiated as part of the
transformation process, it is critical that the newly created models obey implicit and
explicit constraints of the target modeling language, and that only features that obey
the declarative rules of the metamodel are found in the target graph.

1.2 Semantically and Structurally Rich Models

To more accurately describe the work done for this thesis it is important to define what
we mean by semantically and structurally rich models. Well-studied domains such
as control systems, signal processing, network analysis, dataflow processing, etc., are
abstracted using a fairly straightforward metamodel. The semantics of constructed
models are given not through the specific type of the elements of a model, but through
the values of specially named *attributes* of a model. For example, a language for a numerical methods domain, may have basic blocks of specific types (e.g., *Primitive* or *Compound*), but the contents of these basic blocks determines whether multiplication, integration, addition, etc., are the operations to be performed on the outputs and inputs. The contained model attributes must also specify the required numerical accuracy, timestep during simulation, numerical solver to use, etc. Omitting or incorrectly specifying these values may lead to an invalid semantic interpretation of the model. The type of modeling structure where identification of elements is largely dependent on the attributes of an element and not just what type of element one that we would classify as semantically and structurally rich.

In general container models have a relatively large quantity of *contained* models and attributes that determine the model semantics when compared to the *interface* for associating the structures with one another. That is, the container object may have only one/two inputs and outputs and have a well-defined behavior. This simple structure can contain through encapsulation 10-15 objects that have their own attributes, contain other objects, or are associated with one another. We say that containers with a structure such as this are *semantically and structurally rich* models.

It is therefore necessary for semantically and structurally rich models created outside the domain-specific user interface to maintain consistency of the model attributes in order for the models to be semantically valid. Unfortunately, a class-diagram based approach to specifying model transformations can be cumbersome if a great many objects must be created, consistently maintained, associated with one another, etc. Further, the model transformation designer must have a significant knowledge of the various semantic interpretations, in order to maintain consistency, or to debug the model transformation once it is designed and implemented.
1.3 Contribution of This Thesis

This paper presents a method and a tool that allows a modeler to take semantically rich domain-specific models, and generate from those models the model transformation rules that can be used to instantiate (or match) such models. The approach guarantees that the components of the rule obey the necessary constraints (by starting with models created in the DSME), and contain only features that obey the declarative rules of the language’s metamodel.

The process also validates the generated rules in the following way: when those rules are executed, the original model (from which the model transformation rules were themselves generated) is produced.

The generated rules may then be reused with high confidence in new model transformations. If a designer requires other domain-specific structures to be part of a new rule, those structure can be created in the DSME, and new rules may be generated for use.

This paper extends previous work [23] by the addition of formalization for the transformation process, and by explicitly considering the generated rules to be self-validating.
Chapter 2
BACKGROUND

The foundations of model-based design, domain-specific modeling, and various graph and model transformation approaches, are needed in order to understand the contribution of this work. The purpose of the detailed definitions is to provide a convenient mechanism with which the presented work can be compared to that found in the literature.

2.1 Model-based Design

The Unified Modeling Language \cite{54} is a common and widely understand formalism for describing software and system design features. Although the semantics of UML are not agreed upon among experts \cite{22, 4, 41}, the concept of UML class diagrams are related to the presented work, and it is important to understand the concepts of UML class and object diagrams.

A class diagram is a UML model that describes the structure of various software classes, and uses textual annotations to describe the methods and fields of software classes used in a software system. An example class diagram is shown in Figure \ref{fig:example_class_diagram}. In this diagram, the rectangular boxes denote a software class, and the associations between the boxes indicate runtime “links” that can exist between those classes.

![Figure 2.1](image_url) An example class diagram, relating \texttt{Laptop} and \texttt{ExternalKeyboard} classes.
An object diagram is a UML model that describes instances of software classes during system runtime. The object instances are constrained to have the same fields of the type that they instantiate. Object diagrams represent the system’s state at a particular time value; that is, there is no dynamics associated with an object diagram—only instantaneous values can be seen. An example object diagram, where objects are instances of the classes shown in Figure 2.1 is shown in Figure 2.2.

![Object Diagram Example](image)

**Figure 2.2.** An example object diagram, relating instances of the **Laptop** and **ExternalKeyboard** classes.

### 2.2 Domain-Specific Modeling and Domain-Specific Modeling Environments

Domain-Specific concepts play a significant role in almost all successful computer oriented projects. The importance of visualizing ideas through model creation is continually increasing due to increasing complexity of project design. Domain-specific modeling languages provide users with an intuitive method for designing and implementing a system within the set rules of a specific domain. Through domain-specific modeling, all contributors of a project can more easily share ideas through a common known language [27, 46, 33]. Though Domain-specific modeling can be used any field or project, some of the more popular uses include control systems, signal processing, network analysis, dataflow processing and other systems. All of the mentioned systems have rules and structures which need to be held true. These rules and structures
are defined in a created metamodel for the specific domain. With the metamodel in place contributors can then create models which follow there specific rules of their system.

A domain-specific modeling environment (DSME) is the modeling language coupled with various semantic interpretations of models built in that language, and constraints on how models must be created. Many DSMEs automatically annotate created models with information that is semantically relevant for execution, simulation, serialization, versioning, etc. The resulting tool successfully uses encapsulation through containment to hide details that are distributed throughout a model, but semantically relevant.

In order to examine the role of model transformations, some foundations of domain-specific languages must be provided; these definitions are taken from [48].

Let \( T \) be a (finite) set of types, with \( t \) as the element name, and \( T \) be a (finite) set of terms used to build a model, with \( \tau \) as the element name. We use the symbol \( A \) as the abstract syntax of a language (with \( a \) as an element), which in this paper we consider as a finite set of constraints on the construction of a potentially infinite set of models.

We abbreviate the abstract syntax tree as AST, and assume it is partially ordered (to guarantee determinicity of the model). We use \( A_C \) to refer to the concrete syntax of a language, which specifies the appearance of types and terms during model construction. Outside the scope of our consideration is the process of producing the AST from terms expressed using the concrete syntax.

**Assumption 1** (Semantics-free Concrete Syntax). *No semantics is interpreted from the \( A_C \).*

**Definition 1** (Constraints). *Let constraints be defined as \( C = \{c_1, \ldots \} \), a finite set, where each \( c_i : a_j \rightarrow \mathbb{B} \) with \( a_j \in A \). Generally speaking, constraints tell whether a valid syntax structure is in fact semantically invalid.*
There is a subtle difference between constraints and abstract syntax. Other formalisms (see [26]) include the constraint set as part of the syntactic specification of a language, since both apply to the structural forms of the language. While this is certainly true, abstract syntax better expresses allowed structure, while constraints better express contextually required or un-allowed structure. Thus, for example, we prefer the use of a constraint to express cardinalities such as 1..* (especially when only in certain contexts), while abstract syntax expresses cardinalities such as 0..1. Constraints which require association (or containment) cardinalities of 1..* are called constructive constraints in this thesis.

**Definition 2 (Metamodel).** The metamodel is defined as a model of syntax and constraints, formalized as $\mathcal{M}^+ = \langle \mathcal{A}_C, \mathcal{A}, \mathcal{C} \rangle$.

Note that we depart from the convention laid forth in [26] to consider the constraints as a part of the $\mathcal{A}$ tuple. This helps to express various subcases of domain evolution. The semantic interpretation of $\mathcal{M}^+$ is the synthesis of $\mathcal{A}$ and $\mathcal{C}$ through the definition of types, $\mathcal{T}$, and the allowed relations between members of $\mathcal{T}$ (see [39]).

We occasionally refer to the metamodel as a shorthand for a language without semantics, or the subset of a language without consideration of semantic attachment.

**Definition 3 (Language).** Let a language, $\mathcal{L} = \langle \mathcal{M}^+, \mathcal{S} \rangle$. To maintain generality, we consider that $\mathcal{S} = \{ \mathcal{S}_1, \mathcal{S}_2, \ldots \}$, a finite set. This means that a language can consist of a set of several semantic interpretations, which are not necessarily coupled. Usually, the cardinality of $\mathcal{S}$ is 1, but never 0, implying that a language does not exist without semantics. It is appropriate that many examples in the literature refer to semantic mapping as “model interpretation” given that the models may be interpreted in more than one way. To motivate our explicit mention of multiple interpretation, consider a model of a sensor network, that may be interpreted in one semantics for simulation [15], and by another semantic definition to perform design-space optimization [28].
When the semantics are not defined or immaterial, we use the term metamodel. Some publications in the literature refer to a language as a paradigm (see e.g., [39, 25]).

**Definition 4** (Model). Let \( m \) be a model, an instance of some type, \( t \), in a language, \( L \). In this paper, we presume that the relationships between any two models obey the abstract syntax, \( A \), of the language.

We use \( \mathcal{M} \) to represent a model database, a collection of models according to the definitions of a metamodel, \( \mathcal{M}^+ \). Although a model database may express typed multigraph relationships, it was shown in [39, 25] that it is possible to encode such relationships into a tree, by maintaining relationship information as attributes. We can, without loss of generality, express that \( \mathcal{M} \equiv m_0 \), where \( m_0 \) represents the root model. A model database, \( \mathcal{M} \), is a partially ordered set (i.e., poset) (since it is expressed in an AST). To express a model’s conformance to various formalisms, we define the following expressions.

**Definition 5** (Model Database Conformance). A model database is said to conform to a metamodel, when no constructions violate the syntax rules. We use the relation \( \models : \mathcal{M} \times \mathcal{M}^+ \rightarrow \mathbb{B} \). Thus, by \( \mathcal{M}_1 \models \mathcal{M}_0^+ \), we mean “Model \( \mathcal{M}_1 \) conforms to the syntax rules expressed by metamodel \( \mathcal{M}_0^+ \).” Because model conformance is dependent on the syntax defined by a metamodel, we might also say that \( \mathcal{M}_1 \models \mathcal{A}_0 \), to stress that the \( \mathcal{M}_1 \) does not violate any syntax rules defined in \( \mathcal{A}_0 \).

**Definition 6** (Model Database Well-Formedness). A model database, \( \mathcal{M} \), is said to be well-formed in a metamodel, \( \mathcal{M}^+ \), if \( (\mathcal{M} \models \mathcal{M}^+) \land (\forall c \in C, m \in \mathcal{M}, c(m) = \text{true}) \). We use the relation \( \models \mid \models \) as a shorthand, and define it as \( \models \mid : \mathcal{M} \times \mathcal{M}^+ \rightarrow \mathbb{B} \). Thus, to say that \( \mathcal{M}_1 \models \mathcal{M}_0^+ \), we mean that “Model \( \mathcal{M}_1 \) is well-formed to the syntax rules and constraints of \( \mathcal{M}_0^+ \).”

By definition, \( \mathcal{M} \models \mathcal{M}^+ \Rightarrow \mathcal{M} \models \mathcal{M}^+ \).
2.3 Graph and Model Transformation

Model and graph transformations are powerful techniques which have been implemented in a multitude of situations to enhance the functionality of systems some examples include [34] [40] [3] and [6] [5]. [11] further details the importance and utility of graph and model transformations. The wide spectrum of research done on model transformations and its uses are a testament to its importance and utility. Siikarla [45] focuses on the importance of a model transformation and introduced an approach to rapidly developing these transformations. Widespread application can be seen in the base tools and languages that have been created for model transformations in recent years, including developments in algorithm improvement and speedup as well. Bergmann’s [9] and Aßmann [5] work are among the many examples of this line of development. The wider use of model transformations by modelers who are not versed in the subtleties of model transformation tool development motivates the work in this paper, which aims to streamline the development process by demonstrating how patterns can be correctly created.

When discussing model transformation languages it is necessary to address the different approaches that transformation languages take. Two of these approaches are textual declarative rule and visual transformation languages. Atlas Transformation Language (ATL) [24] is one of the languages that falls under declarative textual rules. In these textual based languages the users create textual rules to define a pattern, and these rules are executed on a given source model. Much of the work done thus far focuses on the visual aspects of models, and as such a focus on visual transformation is currently needed. This means that the approach discussed for transformation is only in the theory stages for textual languages like ATL.

Among visual model transformation languages are FUJABA [38], BOTL [13], VIATRA2 [16], VMTS [32], and GReAT [1]. The work presented in this paper is not limited to any specific transformation language and as such could be applied to a wide
range of visual transformation languages. The transformation simplification concepts
can be applied to different language because it takes advantage of the abstraction
seen in most transformation languages and helps the user deal the abstraction in
a simpler form.

For the next subsections the following model transformation definitions are nec-
essary to explain the algorithms an approaches take for the simplification of model
transformations. The relationship between many of these elements can be seen in
Figure 1.1(a)

2.3.1 Model Transformation by Example

A technique for model transformation creation is discussed [53] [52] [8] [49]. This
method of generating transformation rules through examples know as Model Trans-
formation By Example (MTBE). Transformation by example reflects the structure
of other “by example” methods. These methods can be seen in macro generators for
word processors and programming environments. In these cases code is generated
through recording keystrokes from the user. The transformations created using the
transformation by example method are created in a very similar fashion: the user
provided example source and target model along with a mapping model, and the
environment determines how the change can be implemented through model trans-
formations.

For MTBE a user creates target and source models in the respective meta-models
and connects elements from source model to target model to indicate which elements
in the source model correspond to elements in the target model. These models are
created in the transformation environment provided by the transformation engine.
Once the user provides the source model, the target model and the mapping model,
an analysis of these models is done to infer the rules of transformation. There are
a wide range of analysis methods which have been discussed. One of these methods
is rational concept analysis [18]. Another is inductive logic programing seen in [52]. Regardless of the analysis approach the end goal is to infer a pattern from the users provided models. All analysis approaches have the advantage of being able to work in the background making full knowledge of transformation languages unnecessary. One of the limitations of MTBE is that attributes are ignored in mappings because only objects can be mapped to each other. Additionally there is no gauge on the accuracy of the created inference. The inferred transformation found through any analysis is not guaranteed to be the transformation the user desired.

The work most closely related to that discussed in this paper comes from other approaches to transformation simplification, especially transformation by example. The approach of this paper can be distinguished from that interesting, yet different, concept that was introduced and thoroughly explained by Wimmer et al. in [55]. García-Magariño [19] further explains and proposes improvements to the approach and it has been demonstrated by Sun et al. in [50].

The difference of the proposed work to transformation by example is most simply that transformation by example generates a transformation based on the various editing operations made by a user when interacting with the model, while the approach in this paper generates a set of reusable transformation rules from a static model. The results of the work in this paper are intended to give rapid access to complex patterns that emerge from domain-specific types, in order that these low-level patterns may be applied to more complex transformations. Model transformation by example aims to raise the level of abstraction for specifying model transformations, which is an interesting—but different—objective.

It may be argued that the intermediate results of model transformation by example are in fact transformations that could be edited and reused. However, existing research focuses on transforming object diagrams, not domain-specific models, and thus the level of complexity for the source artifact is lower than that supported in this paper. Further, the work in this paper specifically decomposes the creation of the
input models into a decomposed collection of rules in order to aid reusability of the transformation elements. This novel feature again distinguishes the approach from transformation by example.

2.3.2 Model Transformation by Demonstration

Model Transformation by Demonstration (MTBD) seen in [51] [30], is an approach to simplification of model transformation. The target group for transformation by demonstration are users who are unfamiliar with any specific transformation language. Transformation by demonstration takes a sample source model and allows the user to manually transform the model. While the user is demonstrating the desired transformation a recording engine is logging the changes the user makes to the source model. When the transformation is complete the changes recorded are optimized and the pattern is then inferred from these changes. Once a pattern is created pattern matching for the precondition is done and the operations are repeated for every instance of the precondition that is found. [51] specifically uses the tool MT-Scribe which is an Eclipse plug-in for GEM to implement the model transformation by demonstration approach. It is important to note that MTBD is currently implemented for endogenous model transformations and that the final outcome is a transformation pattern which includes both the precondition model and the transformation.

The steps taken to perform model transformation by demonstration can be seen in Figure 2.3. The first step involves the user and the recording engine. The user takes an example model in the source domain and performs changes which adhere to the list of allowed operations. The list of allowed operations are as follows:

- Add an Element
- Remove an Element
- Modify an Element
Figure 2.3. The necessary steps for performing model transformation by demonstration seen in [51].
• Add a Connection

• Remove a Connection

• Modify a Connection

Adding and removing elements specifically refers to structures or objects allowed in the source metamodel. Modifying and element allows the user to change the properties of a given object or structure. An example would be changing the Name if it is a given property in the metamodel. The adding and removal of connections refers to relationships within the model between different structures or objects. The modification of a connection allows the user to change elements like the source, destination and cardinalities of these connections. As the user is change in the example model the recording engine makes a list of operations preformed during the user demonstration.

When the list of desired operations is complete (e.i. the user finishes demonstration) the optimization process begins. Optimization is preformed to remove unnecessary operations from the original list of operation. Unnecessary operations are simply ones that have an identical inverse operation within the set of operations. An example would be adding an element and later removing that exact element. The optimization is a simple search algorithm which traverses through the list of operations provided by the user. At each operation the type of operation and the element involved are both identified. Then all subsequent operations are checked for find inverse operations involving the same element. If an inverse if found both operations are removed if not the search proceeds to the next operation on the list. The optimization helps not only the accuracy of the inferred transformation but also helps improve the efficacy of the transformation for larger models. These advantages and the simplicity of the optimization process make it an important part of process as a whole.

The final list of elements post optimization is then used to infer the transformation. The output of the inference step is a transformation pattern. A pattern differs form a rule in that it includes both the preconditions of the transformation and the
desired actions. The actions are taken from the optimized list of operations. The
preconditions are extracted using the list of all elements and connections from
the source model, the meta-information and the recorded list of operations. The
final steps are optional and can be called on by the user. Precondition matching
can be done on a different model to perform the first necessary step in recreating
the transformation. The transformation can then be performed on the model if the
preconditions were found within the model. The last two steps can also be performed
on the original source model to check for inference correctness.

The difference of the proposed work to transformation by demonstration is most
simply that transformation by demonstration generates a transformation pattern
based on the various editing operations made by a user when interacting with the
model, while the approach in this thesis generates a set of reusable transformation
rules from a static model. The results of the work in this thesis are intended to
give rapid access to complex patterns that emerge from domain-specific types, in or-
der that these low-level patterns may be applied to more complex transformations.
Model transformation by demonstration aims create a simplified environment for
novice model transformation user where as our method focuses more on reducing the
time needed for transformation model creation while still preserving the complexity
and flexibility that transformation languages gives to an experienced user.

2.4 Graph Rewriting and Transformation (GReAT)

The work for this thesis was preformed in GReAT as such it is important to include
a basic introduction to the graph transformation environment. GReAT is a graphical
language used to formalized transformations between various domain-specific mod-
eling languages (DSMLs). GReAT arose out of a shortage of non-textual model
transformation, languages. When using textual languages to define the intended be-
havior of transformation between DSML’s the construction can be time consuming
and costly. The principle design idea of GReAT is an analogous relationship between the behaviors of models and graphs. Models can be seen as labeled multigraph. Looking at models as graphs allows the creator to utilized established graph transformation techniques to create a visual model transformation language. Some of these graph transformation techniques can be seen in [21][42][11][12] and [20].

The the transformation language designed for GReAT is divided into three different sub-languages. These sub-languages are the pattern specification language, the transformation rule language, and the control flow language. All of these languages work to make up the parts of GReAT which are required for model transformation. The language is created in such a way that allows the users to specify class types of domain with generic types in GReAT. One example of this could be with a flow chart which requires arrow connections between elements. The connection class would have a specific name in that language however in the GReAT language it would simply be an \textit{connection} and the characteristic would be specified the the user. This generic structure is utilized in GReAT to help account for the many different types of classes which could be found in both the source and target metamodels.

\subsection{Configuring Transformations in GReAT}

When first creating a transformation in GReAT a user must configure the transformation. The configuration stage allows the GReAT engine to extract information form the proper locations. This is the location where the user inputs the source model, target model, the respective metamodels and sends the files to the first transformation rule. Figure \ref{fig:example} shows and example configuration model and the different elements the model can contain. The SF2FSF\_MetaInfo is an element of type MetaInfo which contains the name and directories of the files which contain the metamodel of the source and target models. OpenSF and SaveFSF are both of type File and contain the location and name of the input and output models respectively. Each of these
Figure 2.4. Example configuration model for a transformation in GReAT from [1].

Figure 2.5. An Example of multiple rules making up a transformation in GReAT from [1].

has an association with an object of FileType which contains the information about what type of model the file is going to be. Finally these the models are connected to the first rule to be executed. The specifics on how a rule is defined will follow the transformation configuration step. The final element is a NoPrompt optional element which simply gives runtime options for different prompts which appear. Most of the information for this section is found in [2], [7], [1] and these references provide more details on GReAT.

2.4.2 Elements of A Rule in GReAT

One of the most important and complex elements in the GReAT language is the Rule. A transformation can consist of one or more connected rules. It is a the users discretion how these rules designed to divide the different parts of the transformation.
Rules are made of pattern specifications with a few additional elements to give the user different options for executing the rule. The rule can do three different things with the specified patterns created within the rule. These three actions are all that is needed for a complete transformation. As such a single rule is enough to specify a transformation, though the single rule limits what the transformation can do. Most transformations are the result of multiple rules connected to one another, as can be seen in Figure 2.5.

A rule in GReAT is made up of seven different key elements. The first of these is a pattern defined as a graph with pattern vertices and edges. Actions are used to map each of the pattern vertices to a specific action. There are three options for actions Bind, CreateNew, Delete. Input Interface allows the rule to receive objects for a previous executed rule. Output Interface allows the rule to pass objects from
the current rule to the next rule. *Guard* gives the option of additional conditions to execute the rule, if no guard is given execution will happen for every matched pattern. *Attribute Mapping* contains code which will be executed for the execution of all the rules. Finally a *Match Condition* us a flag which determine whether the rule is executed on every match or just a single match, it is important to note that there is no way of knowing which exact match it will execute on this selection is arbitrary. With these elements in mind it is useful to show an example of the inner components of a rule.

Figure 2.6 shows gives an example rule with most of the elements a rule can contain. House, Room1, Room2, AdjacentTo, PurchaseOrder, OrderItem are all part of the specified pattern. The actions can be seen in two different ways. In the GReAT user environment an object in the specified pattern will appear a different color depending on the action. Black for Bind, red for Delete and blue for CreateNew. In addition a mark will appear on the lower right side of the element like the one seen in OrderItem. In OrderItem it is a check mark for CreateNew, no mark appears for elements whose action is Bind. If the action is delete an “x” will appear instead of a check mark. Input Interface can be seen in the two input ports one points to House the other points to PurchaseOrder. This link indicates that both House and PurchaseOrder were brought in from the previous rule to be used in this rule. The Output Interface is demonstrated in the two output ports. House goes into one of them and OrderItem goes into the other. This connection indicates that both House and OrderItem will be elements that get passed on to the next rule in the sequence. **HasDoor** is a guard while it is not possible to see what specific code was written from the name that is on the guard it can be assumed that the guard is looking at the boolean value of **HasDoor** in **adjacentTo**. **AttributeMapping** as mentioned before is code which is executed when an rule is executed. While we cannot see the code of **AttributeMapping** this particular Mapping sets the values for the created element **OrderItem** these parameters are **Entity** and **Quantity**. The example does not provide
a Match Condition when no Match Condition is provided it defaults to all matches are executed.

2.4.3 GReAT Actions

One of the most intricate elements in the GReAT language are Actions. Actions allow the GReAT language to define all parts of a transformation in one pattern definition. Many of the common approaches to model transformation have a LHS and RHS division to them. LHS refers to the Left Hand Side and this is generally specified as a separate graph to the Right Hand Side (RHS). Transformation languages in general use the LHS to distinguish when a transformation is to be preformed. A LHS pattern is defined in the transformation and later the engine searches given source models for that specified pattern. When a LHS pattern is identified then the RHS pattern is utilized. The RHS pattern is created by the user to define the changes which are to be implemented on the found LHS pattern. Among the changes that can be made are creating new objects and deleting objects. Defining the roles for RHS and LHS is the main purpose of the Action element in a rule. The Bound objects act to create a LHS and the CreateNew and Delete objects are the RHS. This Action definition makes it so the user only has to create one model and add actions to define the pattern to be searched and the changes that are made once the pattern if found.

2.4.4 Rule Alternatives in GReAT

Rules in GReAT are the building blocks of transformation. However GReAT does provide other elements which can be placed within a sequence of rules to allow for variation of behavior. To begin it is important to define the way in which rules are executed. When a source model is sent to a transformation it is passed to the first rule in the sequence. The rule finds all passing instances of the pattern for a particular source model. The list of elements are then executed and an output for the rule
is produced. When all patterns found are executed they move on to the next rule. This means nothing reaches the subsequent rule before all searching is done in the first rule. In many of the simple transformation cases rules and the execution of these rules are all that is needed to perform the transformation as desired. There are provided alternative elements which can be included in the rule sequence and can be seen in Figure 2.7. The first element Figure 2.7(a) is just a generic rule whose behavior has already been described. One advantage of using a Test Case, seen in Figure 2.7(b), is the ability to run concurrent rules. The test case takes in possible situations or cases and fires when these cases are met. The output can then be sent to different rules and these outputs will be executed concurrently. The use of a For Block, seen in Figure 2.7(c), allows the engine to execute single packets at a time across all rules within the For Block. This means that if a rule has fired multiple times each execution would go pass the next rule before a single one can reach the rule after the next. If instead the rule execution enters a For Block it would get executed across all rules in the block before the next one is execution is passed. This type of execution is
Figure 2.8. Figure 2.8(a) and Figure 2.8(b) show examples of Block diagrams from [1].
advantageous when the sequencing of a transformation takes into account changes which have already been made by the transformation.

Finally the Block, seen in Figure 2.7(d), has many advantages; among these are encapsulation to reduce the appeared complexity. If input and output stay sequential, a block functions exactly like a sequence of rules. An example of this is seen in Figure 2.8(a). However if the output to the block is taken before all the rules are executed, as seen in Figure 2.8(b), the block functions as a delay. In block Figure 2.8(b), the output execution will not continue until all other rules within the block are executed. This delay is at times necessary because the order at which rules are executed is not known when rules begin to break off into different sequences like the transformation for this thesis. The block is the primary alternative used in the work for this thesis.
3.1 Problem

Some DSMEs utilize a fairly simplistic metamodel, in order to support backwards compatibility with various versions of the tool. This approach to developing DSMEs prevents the need to make changes to the metamodel when new types are added, and reduces the requirement for model evolution (as all types in a model can be found in the metamodel). As new types are added to the modeling environment, their structure is created or updated by the DSME’s editing tools to maintain correctness. Approaches such as this are common in tools where simulation is a valid execution of a model, such as Simulink and LabView.

Consider the metamodel shown in Figure 3.1, created in the Generic Modeling Environment (GME) [31]. This language has few types, and generally relies on attribute values to determine the semantics of a particular Compound, Primitive, or Port. Such a DSME may be used as part of a tool-integration framework (in this specific case, permitting an open interface to Simulink models). Clearly there are other designs for specifying these types, but many language designers choose to make the number of types as small as possible for the purposes of opening any model in any version of the tool (even if those models cannot be executed in that version).

Figure 3.2, although not in the concrete syntax of any specific tool, demonstrates how for the domain of signals and systems, several models are associated through module interconnection for a clean and simple diagram. However, the internal structure of each model consists of significant Attribute objects that determine the semantics of each individual Block. In this case, both Integrator and Product are of kind Block,
Figure 3.1. Simple metamodel, demonstrating the declarative nature of semantically and structurally rich models. Stereotypes in the language come from types in GME [31].
so any model transformation that wants to search for *Product* models must look to the structure of a *Block* to find such models.

![Diagram](image_url)

**Figure 3.2.** Instance model of the metamodel shown in Figure 3.1. The internal contents of the *Product* and *Integrator* blocks are shown. Each has a collection of *Attribute* models (squares) and contains other *Block* models (rectangles).

In terms of endogenous transformations, elements of the metamodel (*Block*, *Port*, *Attribute*, etc.) will be used as elements of the model transformation rules. Since the metamodel is relatively simple, the names and values of *Attribute* objects, and
their presence or absence, become the key discriminators for distinguishing among the semantics of a model.

Figure 3.3. A element from Figure 3.1 with internal elements shown.

As an example of how a model transformation interprets a model element Figure 3.3 focuses on a constant element from the metamodel in Figure 3.1. Rules that represent these semantically and structurally rich models as part of the model transformation are shown in Figure 3.4 and are expressed using the syntax of GReAT [7]. This rule shows the creation of a Primitive object that represents a Constant value (e.g., an $n \times n$ matrix of doubles) in the domain. As mentioned the rule when not
defined in the transformation language is seen in Figure 3.3. The *Constant* has a value, min/max ranges, sampling information, data type, LockScale, and a FramePeriod. These particular attributes are what distinguishes a *Primitive* of type *Constant* versus the other elements in Figure 3.4. Every one of the objects in Figure 3.4 had a list of attributes or contained objects which all have values that need to be set.

Omitting even one of these contained objects might invalidate a simulation of the model. Because of all the contained objects, the model transformation for the constant becomes a very large model compared to the one in the original domain. As can be seen in Figure 3.4 there needs to be an AttributeMapping for every element that is found inside the constant along with the constant its self. Figure 3.5 shows the content of two of the AttributeMappings found in Figure 3.4. Figure 3.5(a) shows what coding has to go into an AttributeMapping for a *Primitive* and Figure 3.5(b) shows what code has to go into an AttributeMapping of type *Attribute*. This is just a small subset of all the attributes seen in Figure 3.4. Considering this with the fact that the Figure 3.4 is a representation of the simplest element in Figure 3.2 it an be seen that work needed to create a model in a domain like the one see in the metamodel in Figure 3.1 is significant. In addition to the time consuming work needed to create the models without an ability to consistently—and verifiably—create models in the domain, debugging any transformations of significant size would require significant resources.

This work addresses precisely this issue: how to utilize features of a DSME that promote ease of use, in order to reduce the complexity of writing model transformations for that language. In short, the work enables the development of model transformations to be more domain-specific.
Figure 3.4. A rule to instantiate a structurally correct \textit{Constant} in Figure 3.1 modeled using the GReAT model transformation language. Stereotypes in the diagram for Folder, Model, etc., are according to the syntax of GReAT.
Figure 3.5. Code needed for two different AttributeMappings: (a) is associated with the element constant and (b) is associated with the parameter FramePeriod both of which can be seen in Figure 3.4.
3.2 Approach

The research approach is to begin with domain-specific models that are expected to be used as part of a model transformation. These models may either be the left hand side of a rule (portions of the model that are “matched”), or the right hand side of a rule (portions of the model that are “created” or “modified”). Given these models, the approach generates models in the language of GReAT that can be used as part of either that left-hand or right-hand sides of a rule.

3.2.1 Artifacts

This paper utilizes the GReAT language for model transformations, but the approach is not limited to only GReAT models. However, for the remainder of this paper the terminology of GReAT is used. Although the formalism of category theory may provide a more elegant summary of the artifacts described in this section, none of the shortcomings of set theory are encountered in the following formalization.

Since this paper is concerned with endogeneous transformations, let \( g : M_L \rightarrow M_L \) represent the model transformation function \( g \). A particular model, \( m \in M_L \), conforms to the metamodel \( L \) (i.e., \( m \models L \)), and \( M_L \) is the (infinite) set of models that could be created using the language defined by metamodel \( L \).

A model transformation is made up of a sequence of rules, which fire based on matches of the model to the left-hand side of the rule. Each rule is itself a model, \( r \), which is expressed according to the metamodel of GReAT, denoted as \( L_G \). A transformation, \( g(\cdot) \), is made up of a set of rules, \( \{r_0, r_1, \ldots\} \models L_G \) that may be explicitly ordered to define \( g(\cdot) \). Each \( r_i \) must obey \( L_G \), but in general \( L_G \) does not constrain the output of a transformation. That is, it is not possible to (in general) prove that if \( m_2 = g(m_1) \) and \( m_1 \models L \), that \( m_2 \models L \).

The goal for this paper is to make it easier for the designer of a transformation to say that \( m_2 \models L \), through the generation of rules for \( g(\cdot) \) from some existing model
The generation of $g(\cdot)$ from $m_0$ is carried out by the transformation defined in this work, namely $h$, where $g = h(m_0), g = \{r_0, r_1, \ldots\} \models LG$. The following basic flow is utilized for this approach:

1. A modeler creates $m_0 \models L$, using a domain-specific modeling language ($L$). This model $m_0$ may be a collection of models, or a single model.

2. The model transformation $h$ operates on $m_0$, producing as its output a new model transformation: $g(\cdot) = h(m_0)$.

3. The output model transformation $g$ is made up of a collection of rules, where each rule corresponds to a model found in $m_0$

4. If $g$ is executed on a new (empty) model using language $L$, then it will create $m_2 = g(\emptyset)$, where $m_2 \models L$ and where $m_2 \equiv m_0$

Using this approach, individual rules $r_i \in g$ can be used in new model transformations, as deemed necessary by the designer of the model transformation.

### 3.2.2 Generation of $g$

The synthesized transformation rules that make up $g$ is a decomposition of the instantiation of each object in the source model ($m_0$) as a standalone model transformation rule. In order to synthesize this transformation, a model transformation synthesizer ($h$) is used.

Figure 3.6 shows the overall structure of $h$ and its inputs. The names of the rules are not important—rather, the importance is the demonstration that the transformation is composed of a set of ordered rules. The current implementation of this work defines $h$ for a particular language $L$, but any model that can be created in $L$ can be used as $m_0$. Many of the rules shown in Figure 3.6 are hierarchically defined, meaning that rules may contain rules, may contain other rules and sets of rules. This
Figure 3.6. This sequence of transformation blocks defines \( h \), and when executed on some model \( m_0 \models L \), will generate \( g \models L_G \) (i.e., \( g \) is a GReAT model, generated by a GReAT model).

is convenient for GReAT designers as it promotes modularity, but for the synthesis of \( g \), hierarchical rules are not used.

Each rule created as part of \( g \) has input and output ports as defined by GReAT for rule composition. The reason for including these ports is to ensure that the generated rules can be used in new transformations. When the generated rule is used in a (non-generated) transformation, the interface to using the rule is to connect “output” ports to “input” ports.

The transformation illustrated in Figure 3.6 follows the algorithm for \( h \) to generate the synthesized transformation \( g \), outlined in Listing 1. This listing collects all existing models in \( m_0 \), and for each model it creates a rule, \( r_i \), that can match (or instantiate) that model. If the model \( m_0 \) is too restrictive for reuse in another transformation, then the transformation designer can remove elements of the left-hand side of the generated rule (i.e., the pattern becomes less restrictive) in order to match more models. Likewise, attribute values may be modified when instantiating models.

What is important about the approach is that a transformation designer can depend on this tool to generate the structure of a rule correctly, and that structure
can be modified per the goals of the designer.

**Generator for the Synthesized Transformation**

\( h: \) the generation of \( g \) (output) from an input model \( m_0 \)

\textbf{const}

- \( M: \) list of all models from input \( m_0 \)
- \( T: \) list of all types the language has, from metamodel \( L \)

\textbf{begin}

- Create new transformation folder \( TF \) (in \( g \));
- for each \( m_i \) in \( M \):
  - Create new transformation rule, \( r_i \);
  - Insert \( r_i \) into \( TF \)
- for each \( t_j \) in \( T \):
  - Search \( m_i \)
    - for each instance of \( t_j \) in \( m_i \)
      - Create new pattern, \( p_i : \) pattern based on \( t_j \) instance found
      - Insert \( p_i \) into \( r_i \)

\textbf{end}

*Listing 1: High-level description of the traversal utilized when synthesizing the output model transformation.*

### 3.2.3 Self-validation of \( g \)

In order to validate the synthesized transformation, \( g \), the model transformation is executed with null input, and the output model is compared to the input model.
\[ g = h(m_0) \]
\[ m_2 = g(\emptyset) \]
\[ m_2 \equiv m_0 \]

If \( m_2 \equiv m_0 \) (i.e., the models are isomorphic), then the synthesized transformation is validated. Isomorphism, like an object equivalence operator in object-oriented languages, cannot be generically defined. Thus, it is up to the experts in the domain to validate that the models are in fact isomorphic. Some domains may, for example, make semantic decisions based on the layout of models, while others do not. If layout is an important feature, then the DSME should be modified such that layout is captured within the semantically and structurally rich models’ contents.

To further demonstrate the implementation of the self validation an example of the constant block is presented with the initial block and the produced output given a null input model. Figure 3.7 shows the initial constant models with the elements of the constant expanded in the parts browser on left. On the right we see the the output of the generated transformation when given a null input as the model. When the part browser is inspected it can be seen that the elements which were present in the first constant block are reproduced in the generated model. The only direct perceived differences are in the containing model name one is constSS the other is testmodel. This different is simply a specification of the transformation. It replicates the elements but not necessarily the containing folder. The other significant difference is that the original model include two additional models ProdSS and TermSS. These models are not include because the user can distinguish which created elements get reproduced and this specific example focused solely on the constant element which has been inspected though out the thesis. This reproduction is a validation that the generated transformation did have the proper language to represent the original constant with all elements define.
Figure 3.7. Comparison of original model versus the produced model with null input
Unfortunately the part browser can not show specific values assigned to the elements in constant. This means looking at just Figure 3.7 can not completely verify a successful match unless the user selects elements individually to inspect the parameters. To further prove the successful match the models can be exported as XML versions. Doing this gives containment information with specific values for different parameters. Figure 3.8, Figure 3.9, and Figure 3.10 all show excerpts of these XML files. Figure 3.8 shows the Constant1 XML representation for the original constant in Figure 3.8(a) and the XML representation for the generated model in Figure 3.8(b). When examining Figure 3.8 it can been seen that apart from the id values and the position values all other values between both files are the same. Figure 3.9 show the Constant1 XML representation of the output port for the original constant in Figure 3.9(a) and the XML representation of the port in the generated model in Figure 3.9(a). Like the Constant excerpt if compared the XML is the same apart for id numbers and position value. This similar pattern applies the XML excerpt in Figure 3.10. This particular XML excerpt is the XML representation of the atom Value. These particular elements where chosen at random to show a subset of the file. The file as a whole follows the same idea that elements in the two XML files are identical apart from id numbers and position values.

Although a validation operator is not fully automated, the process is still considered self-validating, because the necessary artifacts to perform validation are provided. In future work, automation of the validation process (with some assumptions) is planned.
Figure 3.8. Excerpt of the Constant1 XML file of the original constant and the generated constant (a) is the XML from the original constant and (b) is the XML from the generated constant.
Figure 3.9. Excerpt of the Constant1 XML file of output port of the original constant and the generated constant. (a) is the original port and its elements (b) is the generated port and its elements.
Figure 3.10. Excerpt of the Constant1 XML file of the Value atom of the original constant and the generated constant. [a] is the original Value atom and its properties [b] is the generated Value atom with its properties
4.1 Case Study

This section describes a case study, where a Simulink model (using an equivalent DSME) is modified in order to filter certain inputs based on matching input types and other context of the inputs. A new filter, created in Simulink, is inserted into the model, and the structural complexity of that filter is demonstrated. As part of the case study, this rule is applied in a separate transformation,

![Diagram]

Figure 4.1. The source artifact for the case study.

4.1.1 Source model $m_0$

The source model used to generate $g$ is shown in Figure 4.1. The selection of this particular structure is outside is a generic arithmetic structure which can be translated with little difficulty into similar problems of optimization through restructuring, loop unrolling, etc.
4.1.2 The generated transformation: $g$

The transformation generator $h$ produces the model transformation $g$ shown in Figure 4.2. Note that $g$ is a *fully functional* model transformation in GReAT, and requires only minor interaction with the modeler to execute. In brief, the user must associate the execution path with the rule to be executed first (although this could be arbitrarily connected if more than one rule is created, this automation is left to future work). This needed user interaction is illustrated in Figure 4.2. For this example the only provided model was Figure 4.1 and as such the only option for the user to connect is the single generated rule or any rule the user created after the transformation is generated.

Because $m_0$ contained only one model in its root (the container for Figure 4.1), the resulting $g$ is made up of a *single* rule $r_1$ (called GeneratedRule in the figure), which creates all four contained models shown in Figure 4.1 and their contents. Moreover, the attribute settings—gathered from the domain-specific source artifact model—are listed to the right of the pattern, and the correct values are given inside those blocks.

*Validation* Upon execution of $g(\emptyset)$, a model $m_2 \models L$, is produced. Upon comparison, this models is considered isomorphic to $m_0$. This isomorphism comparison is currently done by the modeler, and it does not consider layout/orientation of models as an

![diagram]

Figure 4.2. The transformation $g$, the result of $h(m_0)$ where $m_0$ is found in Figure 4.1. Note that the $h$ creates *all* objects inside the container in one rule. The screenshots are taken from the generated GReAT transformation.
Figure 4.3. The Internal elements and structure of the GeneratedRule from Figure 4.2
important semantic attribute of the model. This may not be the case for all models, as anecdotal evidence suggests that in some rare cases scheduling ties during simulation are broken by layout order. In the current state of the practice, more and more modeling tools are designing equivalence operators for particular languages (in order to facilitate model revision systems), so this problem is likely to be solved for many modeling environments.

![Figure 4.4](image)

**Figure 4.4.** A model comparison between the provided input and the generated output when a null model is given. (a) is the original model and (b) is the generated output.

For this example we validate the output by comparing three different elements of the original model and the generated model. The chosen elements to show a validation of the working model are a direct model output comparison, a GME part browser comparison and a few selected lines form the xml representations of each of these models. Figure 4.4 and Figure 4.5 show the three comparisons chosen. Before the comparison was done a auto layout was done on both models to increase the uniformity between the two. As seen in the validation section in methods the only expected differences between the models are locations and assigned ids. Starting with Figure 4.4 it is apparent that all the elements preserved and the connects are made to the proper port. As a side note Figure 4.4(a) is the same model seen in Figure 4.1 just
Figure 4.5. A GME part browser comparison between the provided input and the generated output when a null model is given. (a) is the original model and (b) is the generated output.
Figure 4.6. A comparison of the xml representations for provided input and the generated output when a null model is given. (a) is the original model and (b) is the generated output with a different layout. Continuing with the validation the next comparison is done in the part browser seen in Figure 4.5. When looking at the expanded sections of the browser there are very few difference. The main difference is the name of the top model. Figure 4.5(a) has a top model of ProdSS while Figure 4.5(b) has a top model name of testmodel. The model name of testmodel is the standard name of a model created in the project and has little baring on the output. The second significant difference are the extra models seen under dataflow for Figure 4.5(a). These are there because the input can contain any number of models the user desires. They do not appear in the generated part browser because the only model chosen to run was the ProdSS. The final part to look at is found in Figure 4.6, the chosen excerpt was from the product block. From examination it is easy to see that the xml files are the same with the exception of the id and the position of the elements. With these checks it validity of the produced model can be verified.

Though the determination of isomorphic relationships is left up to the discretion
of the modeler at this time, an interesting potential future work in autogenerating validators could reduce this burden under certain conditions. For instance, as the transformations are endogenous, simply simulating the output artifact and comparing its results to the source artifact may be sufficient.

4.1.3 Reuse of $r_1$

The compositional definition of GReAT means that a transformation is an ordered set of rules that operate on an input model. Since $g$ produces GReAT models comprised of a set of rules, these rules can be directly used in any other transformation that operates endogenously on the language $L$ (recall that $g$ was generated from $m_0 \models L$), simply by connecting the input and output ports. This idea leads to the conclusion that the generated rule seen in Figure 4.2 can be reused and interchanged in different transformations. To exemplify this a few reuses of the generated rule are going to be created in this section. The first and easiest use for the model is to simply connect the rule in sequence and using to generate the original pattern. This is actually the example seen in the validation example. For that particular example the engine looked for an open subsystem and when it found it the pattern was generated. This direct linking with no alterations is not necessarily reuse. Though the possibility for the uses of direct connection are endless. For example a user could decided that every time a constant appears it gets inserted into the generated model. The uses all depend on what rules come before and which follow.

Reuse 1:

The reuse being discussed is to use the created rule as something more creative than just simple insertion. For the first reuse example the goal is to insert a display element when ever a product is found. This transformation would be useful to anyone
debugging a mathematical system. I could also be useful for visually monitoring the outputs.

The reuse of Figure 4.2 for this example comes in using the model to find a specific pattern instead of to create a pattern. The generated rule is then part of the LHS model instead of the RHS Model. The first step taken is to delete atoms and attribute to help speed up the searching process. Then all the remaining elements are selected and the actions are switched from CreateNew to bound. Finally a guard is created to verify the Product primitive is of type Product and output ports are created to pass on to the new rule. Subsystem will get a port to make sure the new displays are place in the right location. Output Port of product is also given an output because that is where the user will get the information to sent to the display. Figure 4.7 shows the results of the GeneratedRule after the changes are made. This can be compared Figure 4.3. It is important to note that a even novice model transformation user can make the instructed changes in a few minutes.

The next part of the transformation is creating the display item. This part of the transformation will have its own rule. While creating this transformation is a bit more difficult than the changes made in Figure 4.7 it is still a relatively simple task. Figure 4.8 shows the internal structure of the for the rule which creates the display. The primitive is the display with the AttributeMapping to set the values that make it a display. The display needs an input port and that input port connects to the output of the product. Finally the line created and the primitive will be contained in the same subsystem that the product was found in. The final transformation is seen in Figure 4.9.

The final part is testing the transformation. Two different model were selected to test the transformation. One simple model with a single product the other is a much more complex model with multiple products. The results of the first test can be seen in Figure 4.1.3. The input model Figure 4.10(a) has a single product. Figure 4.10(b) is the result as can be seen the desired display item was in fact generated when the
Figure 4.7. End result of GeneratedRule after easy changes are made to reuse
Figure 4.8. Contents of the Rule which will create the displays

Figure 4.9. The final transformation with both rules connected
product was found.

![Diagram](image)

**Figure 4.10.** The transformation results for the transformation seen in Figure 4.9. (a) is the input model and (b) is the output of the transformation.

The second test involves four products blocks. The input model for the second test can be seen in Figure 4.11. The output model can be seen in Figure 4.12. The output show four new display blocks attached to each product block in the model. It can be seen that the transformation was successful in adding the desired display blocks.
Figure 4.11. The input model for the second test of the transformation found in Figure 4.9
Figure 4.12. The output of the transformation found in Figure 4.9 when Figure 4.11 is given as an input.
Chapter 5

Conclusions

This thesis described an approach to automatically synthesizing model transformation patterns from existing domain-specific models whose internal structure represents relatively complex semantics. The results demonstrate that the approach supports rapid redesign of endogenous transformations, based on prototypes built in the domain-specific language. This permits model transformation designers to take advantage of any complex semantics that are encoded in structure of various domain-specific models.

Further, the approach provides modelers with tools to rapidly support update of model transformation algorithms, when a domain-specific modeling environment is upgraded to a new version (adding new features to syntactically and semantically rich models). The process to do this is to simply open the model $m_0$ in the DSME, save it as the newest version, and then regenerate $g$ from $h$, and then take the created rules $r_i$ and copy them into the existing model transformation that uses those rules. This ensures that those model transformations can match, and instantiate, models that are known to be valid in the DSME.

Finally, the tool is self-validating in that the synthesized model transformation $g$, when executed on an empty model, produces the original domain-specific model $m_2 \equiv m_0$ from which $g$ was generated. This additional layer of validation permits model transformation designers to check that endogenous transformations are semantically valid before applying them to more complex models.


5.1 Limitations

A minor limitation of the current implementation requires patterns of the transformation graph to be associated to the metamodel elements that they represent. This limitation is a minor annoyance prior to fielding the transformation, but is a straightforward string-matching task that takes just a few moments to perform.

The work as implemented currently supports only one language, $L$ in which $m_0$ can be expressed. As a proof of concept, the definition of $h$ required some knowledge of $L$, though as discussed next, there are identified steps to make the tool usable for other languages.
Chapter 6

**Future Work**

Future work focuses on refining the tool to streamline development. Recent testing suggests that the automatic association between pattern references and their objects in the metamodel can be largely automated through some simple code changes. This will remove the current limitation of requiring a user to re-associate elements of a rule with the metamodel upon which they are based through name-matching.

Further, additional metamodels are scheduled to be tested with this approach. Although the work discussed in this paper utilizes only one metamodel, there are no domain-specific assumptions in the approach. However, the definition of $h$ is domain-specific, and as new languages are used, the definition of $h$ will need to be updated. There may be some ability to automatically generate $h$ based on the definition of metamodel $L$, which could provide significant reuse of this approach to arbitrary languages.
REFERENCES


