METAMODEL DRIVEN

MODEL MIGRATION

By

Jonathan Mark Sprinkle

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

August, 2003

Nashville, Tennessee

Approved: Date:

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

Mary Margaret,

I love you, and I'm proud of you too.
Thanks for being here for me.

Jon

ACKNOWLEDGEMENTS

I give many thanks to my advisor, Dr. Gabor Karsai for being the Best All-Around Advisor™. Gabor, without your excellent teaching skills and motivational abilities, I would not be in the position I am today. Vanderbilt is lucky to have you, as will be any other student under your tutelage.

I also thank very heartily the other members of my committee. Dr. Janos Sztipanovits, for his political insight (and vision for my future career); Dr. Akos Ledeczi, for holding my feet to the fire when it comes to sticking up for the value of my research, and also social interactions within ISIS; Dr. Greg Nordstrom, for (as usual) providing valuable comments in the discussion of all things metamodeling related, not to mention being an all-around good guy to bounce ideas all-around with; and of course Dr. Doug Schmidt, for asking the hard questions, and making me go further than I wanted to.

I also want to thank my double-e professors from my undergraduate days at Tennessee Tech. Those were some good times. Good times.

Most importantly, thanks to my parents, Kenneth and Teresa, for never allowing me to think that something cannot be done; and to my substitute parents, Rick and Cindy, who served as honorary Sprinkles when asking those everyday questions that parents should ask, like, “when are you going to finish?” And most importantly, special thanks to Mary Margaret, for whose love and support I am most indebted.

This research was performed under the sponsorship of the Defense Advanced Research Projects Agency, Information Exploitation Office, Model-Based Integration of Embedded Systems project, under contract number #F30602-00-1-0580, and also the NSF ITR on "Foundations of Hybrid and Embedded Software Systems".

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF DEFINITIONS

LIST OF ABBREVIATIONS

BOTL – The Bidirectional Object Oriented Transformation Language

DEF – Domain Evolution Framework

DSME – Domain-Specific Modeling Environment

GME – Generic Modeling Environment

LHS – left-hand side

OMG – Object Management Group

MDA – Model Driven Architecture

MGA – MultiGraph Architecture

MCL – MultiGraph Constraint Language

MIC – Model-Integrated Computing

MIPS – Model-Integrated Program Synthesis

MM – Model Migration

OCL – Object constraint language

RHS – right-hand side

UDM – Universal Data Model

UML – Unified Modeling Language

XML – Extensible Modeling Language

XSL – Extensible Stylesheet Language

XSLT – XSL Transform

CHAPTER I

INTRODUCTION

One of the disadvantages of evolving a system in terms of its low-level source code is that even a small change in the requirements of the program could necessitate drastic changes in large portions of the code. Consider the Y2k challenge of the late 1990s: the size of the requirement change – change the year representation from two digits to four – was small; but the risk and effort required to implement the change was comparatively large. One approach to mitigating this disparity between code changes and requirements changes is to provide a development environment customized for a system expert that allows the expert to manipulate the system using concepts of the system. The *domain*, or family of systems, has a set of concepts, or ontology, each of which has a particular meaning in the domain; this ontology makes up the set of constructs for a *domain-specific language*. A domain-specific language is used by a domain expert instead of low-level code to create systems in the domain. The domain-specific language decreases construction time and provides a specialized interface for managing the domain concepts. This approach, called Model-Integrated Computing (MIC) [1], can be used to apply a sort of "golden rule" to drive the development of a domain-specific language: *the size of the change in requirements should be proportional to the size of the change in the implementation*. This basic tenet correlates highly with the metric of maintainability for computer programs: the better the rule is satisfied, the easier to maintain the system.

Domain-specific language development, however, is not a trivial task. In addition to the exploration and understanding of the domain, there is the development of the language ontology, abstract and concrete syntax, well-formedness rules and semantics, in addition to their representation and implementation [86]. A technique called *metamodeling* can be used to easily – yet precisely – describe the syntax and static semantics (the well-formedness rules) of a modeling language. The artifact of the metamodeling process – called the metamodel – is generally retained in an object database for later manipulation, and can later be evolved to create a new version of the domain-specific modeling language (DSML) [5].

Domain models are instances of system types, and are used to represent the structural or behavioral aspects of an existing system. The domain models are translated into a *domain artifact* that is useful in a semantic domain. The *semantic domain* is an ontology and semantics that can be used to describe this particular family of systems. Thus, the semantic domain could be low-level code (e.g., C or Java), a processor instruction set, or possibly another domain-specific language. The semantic domain is usually cumbersome compared to a DSML so translators are created to map domain models from a DSML onto the appropriate semantic domain while maintaining the correct semantics of the DSML.

When a system is modified in its instantiation (e.g., if it is modified structurally or behaviorally) then the current domain artifact no longer accurately describes the system. In this case *application evolution* is required. Application evolution is performed through modification of the existing domain models (the models of the system) to reflect the new structure and/or behavior of the application. In this case the "golden rule" of

maintenance is satisfied, since all changes to the system are expressed using the domain-specific language, *not* the implementation language (i.e., semantic domain). The *semantic translator* (i.e., the translator from the modeling language to the semantic domain) performs the task of ensuring that generated application in the semantic domain reflects the intent of the modeler. Application evolution, then, is required by changes to the system, and driven by changes to the model of the system.

The DSML is key in providing the modeler with the ability to perform application evolution in a domain-specific fashion. Before metamodeling became a widely used method for creating DSMLs [4][5][71][77][81][82][84][85][86][87], most such languages were not apt to evolve quickly. One key reason for this was that it was a significant amount of work to modify the syntax of the language, update its semantic translator, and re-educate end-users on the usage of the language. Thus, early DSMLs were usually not fully implemented and deployed until vetted by domain experts. While this resulted in a language that was satisfactory for system modeling, the process of language development was somewhat unsatisfactory in that the domain experts were not equipped with functioning languages during the testing phase, or if they were, then any changes made took a great deal of time. Metamodeling provided the ability to rapidly develop *and change* the domain-specific language. This enabled language developers to provide rapid prototypes of the language, and thus provide a more realistic language for use by domain experts during the vetting process. This also enabled the deployment of language prototypes before they had been vetted by teams of domain experts, with the expectation that the testing and feedback would be provided over a short period of usage and time. However, this ability to rapidly create and deploy modeling languages came at a cost:

changes to the language also made obsolete any *domain models* (i.e., models created using the DSML) created before its update.

However, domains can – and do – change in their essential definition over the passage of time. That is, eventually the entire family of systems change in some way, and the entities and principles of the domain that were present in the original design may change or may be removed for reasons outside the control of the language. The evolution of the family of systems is called *domain evolution*. In this case, the DSML – as it is a model of the domain – must be updated to reflect these changes. The system requires maintenance on the metamodel level – resulting in an evolution of the domain-specific modeling language. After changing these formal specifications, the metamodeling translator is used to generate the evolved domain-specific modeling language. Domain evolution, then, is required by changes to the domain, thus it should be driven by changes to the model of the domain – the metamodel.

Successful evolution of the DSML is a necessary – but not sufficient – condition of a successful domain evolution. In fact, domain evolution carries with it ramifications that extend past the metamodel to the domain model, and generally all the way to the semantic domain and application. The valuable portion of an MIC solution is not the modeling language itself, which can be generated, but the body of models that is built up by the modeler. When domain evolution is required the model databases created with the old DSML may no longer be an accurate description of the system they once helped implement, due to the changes required in the semantic translator and/or semantic domain. Drastic changes may need to be made to the models such that they map to the appropriate concepts in the semantic domain. This introduces the following conundrum:

"In the evolved domain, should we assume the intent of the modeler, or the intent of the models?" Either the semantics of the domain models should change with the evolution of the domain (respecting the intent of the modeler), or they should remain the same (respecting the intent of the semantic domain).

In these cases, the domain models may need to be modified to be correct not only syntactically, but also semantically. Regardless of the reason why the changes were required, the objective is the same: for the domain models to be *correct* in the evolved domain. Correct domain models have the appropriate semantics such that they accurately describe the system in the semantic domain – which may also have changed. The extent to which the domain models must be modified (and the algorithm to modify them) is a function of the changes to the DSML and its mapping onto the semantic domain, as well as possible changes to the semantic domain itself. This is discussed further in Chapter III.

The requirement that the domain models be "correct" in the evolved domain may be met by recreating the existing domain models using the evolved domain language. Rebuilding models by hand is a laborious task, and prone to errors. If this method were applied *en masse* it would result in a colossal waste of time and the possibility of undiscovered bugs due to erroneous entry. As the semantic mapping of the DSML is defined in terms of syntax patterns, it is possible to use pattern recognition to transform subgraphs of the existing domain models into the subgraphs in the evolved domain such that their mapping into the semantic domain is semantically correct, thus recreating the existing domain models using a generated transformation, rather than human labor. This

dissertation contends that these transformations can be applied to solve the domain evolution problem.

In order to claim a solution to the domain evolution problem, one key element should hold true: that the "golden rule" of maintenance introduced above is satisfied – i.e., that the size of the change to the domain is proportional to the effort required to evolve the domain models. It is, of course, possible to write software using a low-level or domain-independent language (e.g., Java or C++) that will handle the evolution of a system from one context to another (a compiler is a common example), but there would be a distinct advantage to a tool that could generate this transformation without resorting to low-level coding.

Creating a generic evolution scheme for complex languages that gives significant freedom to the programmer (e.g., memory indirection) or that utilize non-object-oriented techniques for basic program execution (e.g., `goto` statements) is a daunting task. However, the evolution of models created using formal methods is simpler than the evolution of software in general due to the restricted nature of most formal methods used to create languages. Modeling and DSMLs are convenient for creating systems in well-defined domains because of their restrictive nature. More importantly, DSMLs are defined using metamodels; in essence, the metamodel acts as a schema for the domain-specific modeling language. Then, the evolution of a domain-specific modeling language can be performed by creating an algorithm that is composed of elements from the language metamodel. The execution of this algorithm on existing domain models to transform them into domain models that are correct in the evolved domain is called *model migration*.

Currently there does not exist any DSML that is customized for the development of domain model evolution algorithms. Such a DSML would allow metamodelers who choose to evolve a DSML to reuse existing model databases – thus saving a significant cost of human labor. Furthermore, such a DSML that was metamodel-based (i.e., that used primitives that were elements from the original and evolved DSML metamodels) would strengthen the ties of the solution to the metamodels – thus providing the modeler with elements that are most relevant to the algorithm. Finally, such a DSML should allow the description of the evolution algorithm to be independent of whatever method is actually used to modify the physical model database, allowing an algorithm to be applied as a library across different modeling editors and storage formats. Given these requirements and objectives, my thesis is,

A description of the change in semantics between an old and a new DSML is a sufficient specification to transform domain models such that they are correct in the new DSML. Further, the pattern that specifies the proper model migration is driven by the change in semantics, and may be fully specified by a model composed of entities from the old and new metamodels along with an algorithmic description for their modification.

This dissertation details the definition, requirements, and implementation of a DSML customized for the domain of domain evolution. The backgrounds chapter provides some important definitions of modeling science, describes archetypal problems that provide valuable insights into the domain evolution problem, and gives a formal definition of the domain evolution problem. At the end of the chapter there is a review of

the current state of the art of mapping technology (e.g., graph-rewriting) and currently implemented domain translation solutions.

Next, justification for and a description of a domain evolution framework is provided. The model of computation, ontology, and interface for generic extension of the framework are all described in this chapter. In the following chapter, a particular instantiation of the domain evolution framework is examined. The domain evolution tool is aimed at the MetaGME modeling paradigm, and generates XSL graph-transformation specifications, as required by the MetaGME language.

This domain evolution tool is used in the next chapter in a case study, which presents information on how to migrate models using the tool with an example driven by the evolution of a domain in another area of research. The artifacts and results of the case study are examined, and compared with theoretical domain evolution solutions created by hand or with other state of the art tools. Finally, conclusions of the presented research, and recommendations for further investigation are given.

CHAPTER II

BACKGROUNDS

A deeper understanding of the concepts and current research is required before the thesis can be justified. The nomenclature of model-integrated computing is provided in this section, as well as state of the art research of archetypal problems of which model migration is a specific instance. For more information on these research subjects, please see Appendix A.

Model-Integrated Computing

Model-Integrated Computing (MIC) leverages semantics embedded in the concepts of a domain by applying those same semantics to formalized modeling language primitives that correspond to domain concepts. Then a mapping is performed to synthesize program output in the semantic domain that implements the structure and behavior of a system in the domain. The translation of domain-specific models into the semantic domain using MIC techniques is called model-integrated program synthesis (MIPS).

As its own domain, MIC also has concepts that require definition before continuing with the discussion of the background section. Although modeling and metamodeling are extensively studied and referred to in research today, the specific meaning of key terms and phrases can vary between researchers. To avoid confusion, the following definitions provide context for the domain concepts of MIPS.

- **CBS** – Computer Based System. The CBS is modeled by domain models

- **Domain** – the realm of existence of a CBS
- **Domain Model** – model created using a domain-specific modeling language
- **DSME** – domain-specific modeling environment, a programming environment that conforms to a paradigm
- **DSML** – domain-specific modeling language
- **Executable Model** – the artifact that is produced from the interpretation of the domain models; written in the language of the semantic domain
- **Metamodel** – the model of a domain (or more specifically, the model of a domain-specific modeling language). It defines types of models that are valid in a particular domain (the ontology of the DSML) and their legal construction
- **Model** – abstract representation of an entity or system
- **Paradigm** – the tuple consisting of metamodel ontology, syntax, static semantics, and dynamic semantics that makes up a MIPS environment
- **Semantic Domain** – the domain of the output of a MIPS environment. This is typically another language or domain, and is one implementation language of the CBS
- **Dynamic Semantics** – the meaning of models as they exist; dynamic semantics are applied to models during interpretation in order to produce the executable model
- **Static Semantics** – rules that define the well-formedness of a model, but that are not part of the syntax (defined as constraints on parameters and domain model structure)

- **Interpreter** – also known as a *semantic translator*, applies the dynamic semantics of the domain to the domain models (also serves as the synthesizer in the MIPS framework) and produces the executable models that operate in the semantic domain

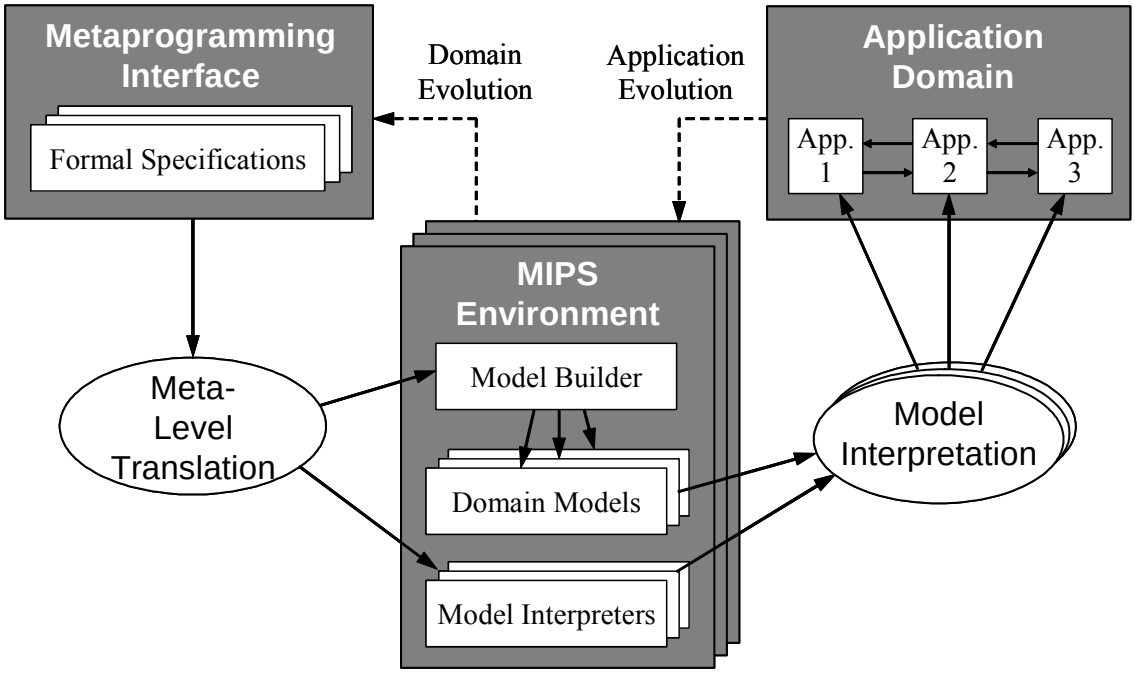


Figure 1. Overview of Model Integrated Program Synthesis (MIPS) [3]

Storage of Domain Models

Domain models are instances of objects defined in a metamodel, and they are the formal description of the CBS [3]. Allowable domain models must be defined within the metamodel, or the model will not be syntactically correct. Also, the construction of domain models is done with the concepts and semantics of the domain in mind, and thus the ontology of the metamodel plays a large role in the selection of domain model types.

The ontology of the metamodel also plays a large role in the persistence format of domain models.

An instance of a domain model contains instance information (true only for that particular instance, such as attribute values, or connection information), as well as meta-information (true for all instances of this model, such as such as how many attributes exist, and what their names/types are). When domain models are loaded into the DSME this meta-information is used to appropriately visualize instances, and also during interpretation for the translation into the semantic domain.

Although at least one argument to the contrary has been presented [90], in general it is a valid assumption that when looking at data, it is possible to obtain information about the type of that data, since that data must be stored in terms of its type somehow. That is, it is always true that the data is useful if its type is known, but this dissertation assumes that the type can be inferred or derived given only the data. This assumption will play an important role in establishing methodologies for specifying model transformations, as discussed in Chapter IV.

DSML

Domain models are created with the ontology of the *domain-specific modeling language* (DSML). This is the language used by the MIPS environment, and is written and customized by a language expert to allow programmers to rapidly create programs. The ontology of a DSML is crafted such that a domain expert is capable of writing programs without a lesson in general-purpose programming. One popular way this can be accomplished is to create a visual language, so that users need not understand primitive language constructs to productively use the language in a relatively short period

of time. It is important to distinguish between a DSML and a domain-specific API. The API is an interface to a library that can be accessed with another language (e.g., a Java jar file or a C++ library) but is not a standalone application. That is, a shell main function that calls the API must still be created. The DSML is free from low-level language requirements that non-programmers find prohibitive. The goal of a DSML is to provide a way for domain experts to leverage the power of the computer without needing to go to programming experts when updates to the system are necessary.

Metamodeling

Metamodeling is the formal definition of the modeling concepts that may be used to define systems within a domain. Modeling concepts are not only the actual domain concepts (e.g., processes in a signal processing domain, or assembly lines in a factory domain) but also standard modeling abstractions – patterns that provide a prototypical solution to a modeling problem – directly supported by the tools. Many such modeling abstractions exist in engineering but are often focused on a particular solution space or sub-domain. A precept of metamodeling is the existence of a core set of fundamental modeling abstractions that (as a set of archetypes) is adequate to express the design concepts, notions, and artifacts used across all semantic domains. Table 1 lists the elements of this set.

Whether each of these abstractions is represented by a first-class concept (i.e., it may be instantiated) is left up to the metamodeling approach used. The Generic Modeling Environment (GME) [100], which is the modeling environment used in this dissertation, is a meta-configurable modeling environment that provides in some form the ability to model using the concepts in Table 1. The metamodeling approach used in

GME promotes some of these abstractions to be first-class concepts, while the remaining abstractions are supported through special embellishments on the basic metamodeling constructs. Other metamodeling tools, such as DoME [84] and MetaEdit+ [85], may choose different members of this set to be first-class objects.

Table 1. Archetypal concepts and their descriptions in metamodeling [87]

Archetypal Concept	**Description**
Classes	Specific classes of entities that exist in a given system or domain. Domain models are entities themselves and may contain other entities. Entities are instances of classes. Classes (thus entities) may have attributes.
Associations	Binary and n-ary associations among classes (and entities).
Specialization	Binary association among classes with IS-A semantics.
Hierarchy	Binary association among classes with "aggregation through containment" semantics. Performs encapsulation and information hiding.
Module interconnection	A specific pattern of relationships among classes. Classes can be associated with each other by connecting their ports (specially marked atomic entities contained in the classes).
Constraints	A binary expression that defines the static semantic correctness of a region of the model: if the objects of the region are "correct," the expression evaluates to "TRUE."
Multiple aspects	Allows partitioning a complex model according to part categories. Used for visibility control, but may also be used for aggregating specific properties of models with respect to specific concerns.

Members of the set of metamodeling archetypes (that is, the ontology of the meta-metamodel) are instantiated in a metamodel to create an instance of an archetypal formalism. For example, the archetypal formalism of *containment* can be instantiated to

formalize that an `Automobile` can contain one or more `Seat` objects. The semantics of the *containment* archetype immediately dictate the relationship between the *parent* (`Automobile`) and *child* (`Seat`). This provides a rapid way in which to specify the abstract syntax of a domain-specific modeling language. The domain-specific modeling language is in turn used to specify the structure and behavior of domain applications [87].

The Meta-metamodel

The DSME's language is represented by a metamodel, and the ontology used to create that metamodel is found in the *meta-metamodel*. The meta-metamodel is the most abstract of all languages in domain-specific programming. Its ontology is the fundamental set of all objects that may be used for creation of domain-specific modeling languages (see Table 1). Interestingly, the meta-metamodel is also self-descriptive – meaning that the creation of the meta-metamodel may be done with the meta-metamodel (e.g., EBNF [4]). Also, there is more than one meta-metamodel. A meta-metamodel can be used to describe DSMEs, and can be used to describe itself, as well as any other meta-metamodel. Any number of meta-metamodels can be used to express the syntax and ontology of a language such as C++, but some are more convenient than others. For the purposes of this paper, we consider that the DSME's language, once evolved, is still expressible using the same meta-metamodel, i.e., we will use a single meta-metamodel.

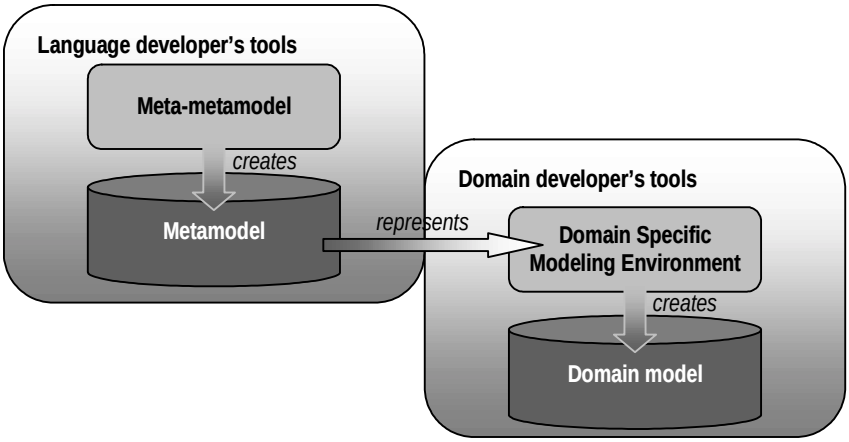


Figure 2. The role of the meta-metamodel as a language developer's tool, and how it related to the tools of a domain developer

Modeler's Intent

Charles Simonyi announced in 1999 that "The Future Is Intentional" [7]. Intentional Programming (IP) [8][9] is an example of a growing trend to build software solutions *by* design rather than *referring to* the design. All programming languages are used to encode of the intention of the programmer (hence the original name of programs, "codes"). Sometimes the intention is obfuscated in difficult to understand syntax (in the case of languages like LISP) or in optimized behavior obtained by low-level instruction (e.g., pointer arithmetic in C). MIC attempts to abstract the details of implementation and focuses instead on the details of the design – depending on the semantic translator for optimization and guarantee of behavior.

As a kind of IP, MIC provides an interface that is most like the design of the final system, and transforms that design into the semantic domain through a translator or *interpreter*. The abstractions of a well-designed domain-specific language are convenient for expressing the *existence* of object instances in a particular domain. The structure and

behavior of domain objects – traditionally specified in the design, and then encoded into the semantic domain – can be immediately translated into the semantic domain if a semantic translator exists for the MIC domain. In this way, the intention of the system is specified by its existence, rather than encoded into a domain-independent language.

Domain Evolution

As with any domain, the power of its domain-specific language is directly tied to the abstraction level of the domain concepts – notably, that the more semantic meaning attached to domain-concepts, the less time the modeler spends specifying the domain models. In the domain evolution domain, there are many definitions and keywords used that enable a terse discussion of the subject material with few pauses to clarify the meaning of context-sensitive words such as "language" and "model". This section should clarify the meaning and reasoning behind the nomenclature used in the discussions of domain evolution.

Driving Forces

A domain-specific modeling paradigm consists of *language syntax*, *ontology*, *static* semantics, and a *semantic translator* (also known as *dynamic semantics*). *Domain models* created using a paradigm are translated into the *semantic domain* with the semantic translator at interpretation time. Changes to any of these key members of the model-driven design process require an evolution of some kind, the two major types being application evolution and domain evolution. Application evolution is outside the scope of this dissertation, which primarily focuses on domain evolution. Domain evolution is required by one of two reasons,

- if any of these members of the paradigm tuple change, or
- if changes take place in the semantic domain.

Regardless of the reason for the change, the *solution* is the same: to modify the domain models such that they are correct in the semantic domain. The modification of the domain models when required by domain evolution is termed *model migration* (MM).

There is a finite, and ordered, set of operations required to solve the domain evolution problem. They are as follows:

1. Recognition of a change in the semantic domain or paradigm
2. Evolution of the ontology, syntax, static semantics, and semantic translator
3. Migration of all models

Examination of the driving forces of domain evolution (e.g., whether by paradigm change or semantic domain change, or both) is crucial to the creation of the algorithm used to migrate the models. This is examined in detail during the case studies in Chapter V.

Migration versus Transformation

Since the metamodel typically undergoes an *evolution* rather than a *revolution* during domain evolution, it is a fair assumption that the metamodel and its evolved self are more similar than they are different. Given this precept, there are significant advantages to pursuing a migration approach over a transformation approach. *Migration* is defined as creating an evolved model database from an existing model database using descriptions of the difference of two sets of ontologies, rather than their similarities. The following formalization defines migration. For a detailed description of the used functional notation, please refer to Appendix B.

Let δ be a domain
Let B be the boolean set

$$B \equiv \{true, false\}$$

Let O be a set of object types (an ontology)
Let Y be a set of syntax rules between objects $o_1, o_2, \ldots o_n \in O$
Let S_δ be a set of semantic mapping functions to a domain, δ,

$$S \equiv \{ \{ s(o) \} : O \rightarrow \{ \delta\text{-object, undefined, false} \} \mid o \in O \}$$

where a δ-object is an artifact in the δ domain.
Let C_δ be a set of static semantics (constraints) for the domain, δ,

$$C \equiv \{ \{ c(o) \} : O \rightarrow B \mid o \in O \}$$

Let α be a paradigm, a quadruple of <O,Y,C,S>
Let α' be the new paradigm, quadruple of <O',Y',C',S'>
Let M be a model database (set of domain models) conforming to paradigm α
Let M' be M evolved to conform to an evolved paradigm, α'
Let m be any original domain model contained in the set M
Let m' be m evolved to conform to a new paradigm α', contained in the set M'

m-function: partial function that operates on a model, m, contained in a model database, M, and produces either the empty set or a model m', which is a member of the set M',

$$\text{m-function} \equiv \{ mig(m) : M \nrightarrow M' \cup \emptyset \}$$

Definition 1

Migration: total function that operates on a model database, M, and produces a model database, M', using a set of m-functions. In the absence of a defined m-function for a model, m, the model is isomorphically copied into the model database, M',

$$\text{Migration} \equiv \{ \{ mig(m) \} : M \rightarrow M' \mid$$

```
∀m∈M
  if m ∈ dom(mig(m))
    m' = mig(m)
  else
    m' = m
  M'.insert(m')
}
```

Definition 2

The formalization states that the absence of an m-function *f*(*m*) implies that *m'* will become an unchanged *m*. However, *m'* can be a modified version of *m* given that an m-function *f*(*m*) is defined. The migration approaches heavily relies on an isomorphism operator (see Appendix C).

Transformation is similar to migration, except in the case that there does not exist an m-function *f*(*m*). Transformation dictates that in this event *m* will not be placed in the output set. Formalized,

```
Transformation: total function that operates on a model database, M,
  and produces a model database, M', using a set of m-functions.  In
  the absence of a defined m-function for a model, m, the model is
  ignored,
    Transformation ≡ { { mig(m) } : M → M' |
                    ∀m∈M
                      if m ∈ dom(mig(m))
                        m' = mig(m)
                      M'.insert(m')
                  }
```

Definition 3

A direct consequence of using a transformation approach is this: that a mapping is required for all objects in the ontology.

This research claims that migration is a superior method of rewriting for the domain evolution problem, since the absence of a function dictates the copy of an isomorphic object to the output set. Given that domain evolution generally evolves models between two similar metamodels, this allows the modeler performing the evolution to specify *only* those differences between the two metamodels, thus maintaining the golden rule of maintenance – that the size of the change in requirements is proportional to the size of the effort required to implement the requirements change.

Semantics versus Syntax

Syntax and semantics are quite often distinguished as two different, yet related, aspects of any language (visual and modeling languages included). While it is not difficult to convince someone that these two concepts are not identical, their relationship to each other during the model evolution solution is not quite as clear-cut. In order to explore this further, let us examine the meaning of a semantic versus a syntactic evolution. The following definitions and formalizations, as well as those introduced in the previous section, apply,

Syntactic Transform: transforms a model, m, in the model database M into an evolved model, m', such that m' is syntactically correct according to the δ syntax rules of a new paradigm

$$Tr_{SYN} \equiv \{ Tr_{SYN}(m): M \rightarrow M' \mid \text{transform m into m'} \}$$

Definition 4

Syntax Evaluator: evaluates a model, m, in the model database M against the set of syntax rules, Y, that exist within paradigm α,

$$Syn \equiv \{ Syn(m,\alpha) : M \rightarrow \{ B \} \mid c(m) \}$$

Definition 5

Semantic Evaluator: executes the set of semantic functions for a model, m, in the model database M, according to the domain semantics, S_δ, of a paradigm α

$$Sem \equiv \{ Sem(m,\alpha) : M \rightarrow \{ \delta\text{-object, undefined, false} \} \mid s(m) \}$$

Definition 6

Static Semantic Evaluator: evaluates the static semantics of a model, m, in the model database M, according to the static semantics C, of the paradigm α,

$$Sem_{STAT} \equiv \{ Sem_{STAT}(m,\alpha) : M \rightarrow \{ B \} \}$$

Definition 7

Semantic Transform: transforms a model, m, in the model database M into a model, m', such that m' is correct according to the 1. δ semantics, 2. δ static semantics, 3. δ syntax rules of a new paradigm, and 4. previous intent of the models or modeler (as appropriate)

$$Tr_{SEM} \equiv \{ Tr_{SEM}(m) : M \rightarrow M' \mid \text{transform m into m'} \}$$

Definition 8

Note that these transforms may be used either by a migration approach or a transformation approach. In fact, the difference between migration and transformation could also be distinguished by their ability to produce default transforms for objects not specified: migration produces an isomorphic transform, and transformation does not.

Syntactic Migration

This is a crude method for the implementation of model migration. This method modifies the existing domain models sufficiently (either through deletion, or a type of

search-replace algorithm) such that the models obey the syntactic rules of the new system paradigm. When this takes place, we say that a *complete syntactic migration* has taken place. A formalization of syntactic migration is as follows:

```
Syntactic Model Migration: transforms a model database, M, into an
  evolved model database, M', such that M' is syntactically correct in
  a new paradigm, α',
   Preconditions:
     ∃m ∈ M s.t.   Syn( m, α' ) == false
     and
     ∀m ∈ M   Syn( m, α ) == true

     MM_SYN ≡ { f(M,α,α') : M → M' |
           ∀m∈M
             if Syn(m,α') == true
               m' = m
             else
               m' = Tr_SYN(m)
             M'.insert(m')
        }
   Postconditions:
     ∀ m' ∈ M'   Syn( m', α' ) == true
```

Definition 9

There are two situations in which this technique is advantageous,

- if the required modifications are type-based, or
- if the system is in development stage, and the old domain models were not intended for the semantic domain

One drawback to a syntactic migration is the complete disregard for the semantics of the domain models. If syntactic migration is used to migrate domain models intended for the semantic domain, then a domain expert must examine the migrated domain models carefully to ensure their semantic correctness. Depending on the number of models modified during the migration, this task may be too large to complete by hand. Another possible problem is error on the part of the domain expert, either in ignoring or

misinterpreting an incorrect domain model, thus leaving a bug to be found at a future time.

Just as a trade-off analysis is required to measure the benefits of creating a modeling paradigm against the time required to develop one, it pays to perform a similar analysis to justify the creation of a syntactic evolution (as opposed to a more complete semantic migration). This decision is based on the amount of changes in each model, the amount of labor anticipated by the domain expert after migration has taken place, and the number of model databases that need to be changed.

Semantic Migration

This is the most sophisticated method for solving the domain evolution problem. Semantic migration requires that the meaning of the old domain models is preserved after the transformation, *and* that the new domain models conform to the entire set of static constraints required in the new paradigm. When these two needs are satisfied, we say that a *complete semantic migration* has taken place. A formal notation of this requirement is as follows:

```
Semantic Model Migration: transforms a model database, M, into an
  evolved model database, M', such that M' is syntactically correct,
  and semantically correct in a new paradigm, α',
   Preconditions:
    ∃m ∈ M s.t.  Sem( m, α ) != Sem( m, α' )
         or
         Sem( m, α' ) is undefined
         or
         Sem_STAT( m, α' ) == false
    and
    ∀m ∈ M   Syn( m, α ) == true

   MM_SEM ≡ { f(M,α,α') : M → M' |
           ∀ m ∈ M
             if S_δ(m,α') == S_δ(m,α)
               m' = m
             else
               m' = Tr_SEM(m)
             M'.insert(m')
     }
   Postconditions:
    ∀ m' ∈ M'   Syn( m', α' ) == true
         and
         Sem_STAT( m', α' ) == true
```

Definition 10

Satisfaction of all of the requirements for semantic migration is not a trivial task. The domain knowledge of the human programmer creating the evolution algorithm must be extensive, as the *meaning* of some syntax in the old domain models must be accurately transformed into the appropriate syntax in the new domain models that will give that same *meaning* in the new domain. This is in stark contrast to a syntactic migration, where out-dated syntaxes can be deleted or restructured, with no thought given to the former or future semantics of the domain models. By closely examining Definition 9 and Definition 10, several things are apparent:

1. the postcondition for syntactic migration is a postcondition for semantic migration (i.e., semantic migration occurs only if syntactic migration occurs),
2. semantic migration has additional constraints that speak of semantics of domain models,
3. syntactic migration does not address semantics at all,

4. no matching of semantics occurs in the postconditions of semantic migration,
5. syntactic migration includes preconditions not found for semantic migration (namely, syntactic mismatch).

Out of these five, the last two items are the most interesting. The first of these – semantics between the domains do not necessarily match – seems somewhat counter-intuitive. After all, why would one perform a semantic model migration if the semantics were not to be preserved? The answer is that it is not always necessary (or even desirable) to preserve the same semantics, but the resulting semantics should always be *correct* in the new domain. Thinking back to the reasons for domain environment evolution, they revolve around evolution of the paradigm versus evolution of the domain. When the paradigm evolves separately from the domain it is generally preferred to preserve exact semantics of the output models *m'*. However, when the domain evolves along with the paradigm, then the semantics of the output models should match the new domain, and should be preserved only as much as the TR_{SEM} deems appropriate.

For the last point, the fact that a syntactic mismatch need not be in place for a semantic migration to take place is, perhaps, perplexing at first. More thought, however, reveals that just because data types match does not mean that data semantics do. Consider, for instance, the well-publicized mistake in the Mars spacecraft that failed based on misunderstanding of the units of given data (meters versus feet) [13]. The semantics of the data were important – but there was not a problem in the syntax (e.g., reading a floating point value).

Since so many factors must be considered, and a great number of syntax patterns might need to be examined, semantic migration requires a more significant time and

training investment to carry through correctly. However, it is generally cheaper to migrate models than to rebuild them. For significant amounts of data models, the reward of developing a transform of old data into useful new data might be worth the cost of creating the semantic evolution algorithm.

Syntax and Semantics in Evolution

Although syntax and semantics are frequently distanced both in definition and discussion, their relationship in the domain evolution process is unique compared to most other software applications, because the syntax and semantics are inputs to the domain evolution process. One interesting property of syntax and semantics in domain evolution is the link between semantic migration and syntactic migration. In fact, semantic migration cannot take place unless a syntactic migration has occurred.

This claim is best justified by a consideration of its negation (i.e., it is possible to perform a complete semantic migration without a complete syntactic migration). If this is true, then there could exist a domain model that, once evolved, is a semantically accurate model in the domain. However, if syntactic migration has not occurred, then there would exist syntactic problems that would prevent the new domain model from being valid in the new DSME. Thus, it is not possible to perform a semantic migration without performing a complete syntactic migration (i.e., a semantic migration subsumes a syntactic migration).

The link between syntax and semantics is further strengthened by the following claim: that if a complete semantic migration is performed, then the process by which the semantic migration is specified must be defined as a syntactic migration. That is, the semantic changes are implemented using syntax transforms. While this might seem

extraordinary, consider how the semantics of a language are defined – in terms of the syntax of that language. The old and new DSMLs are two languages, and the domain model evolution may be cast as a pattern-based transformation, where syntax patterns in the old language are transformed into a semantically accurate new syntax.

Syntax and semantics are decoupled as well as united in a semantic model migration. They are decoupled because the semantic model migration definition is aimed at solving the semantic problem, not the syntax problem, but the syntax problem is also solved. They are united because the syntax patterns are used to specify the semantic migration.

Basic Types of MM

There are five necessary conditions in order that a MM solution is required to reuse domain models after domain evolution, as was discussed in the previous section. One of these five requirements was a change to the metamodel. In terms of the syntax of the metamodel, there are a finite number of cases that will result in the need for model migration.

Depending on the construction of the meta-programmable environment, and the instances of the domain models, certain changes to the paradigm might not require any changes to the domain models for them to be used in the new paradigm. However, since not all meta-programmable environments are created equal, it is necessary to delineate each type of change. Table 2 gives each type of change – as well as whether that type of change will require modification of the domain models – for three meta-programmable environments: DoME [84], MetaEdit+ [85], and GME2000 [83]. In the far column, a

comparable change to a database schema is given to aid in the understanding of the type of metamodel change.

Table 2. Types of changes that require model migration. Rows 10 and 11 show that change requirements are tool-dependent in these cases

Type of metamodel change		Affected domain models (of this type) are present	Change is required [83] [84] [85]	Equivalent schema change (for database)
Additions				
1	Addition of new type **A**	☐	☐	Addition of table **A**
2	Addition of new attribute of type **A**	☑	☐	Addition of column in existing table **A**
3	Addition of association between types **B** and **C**	☑	☐	Addition of column for database key reference between two tables **B** and **C**
4	Addition of type(s) **E** derived from type **D**	☑	☐	Creation of view **E** based on existing table **D**
5	Addition of constraint on type **F**	☑	☑	Addition of database constraint **F**
Deletions				
6	Deletion of an attribute of type **A**	☑	☑	Deletion of column in non-empty table **A**
7	Deletion of an existing type **B**	☑	☑	Deletion of non-empty table **A**
8	Deletion of association between types **D** and **E**	☑	☑	Deletion/Rename of database key in table **D** which is used in table **E**
9	Deletion of constraint on type **F**	☑	☐	Removal of database constraint
Modifications				
10	Renaming type A	☑	☐ ☑	Rename non-empty table **A**
11	Renaming attribute of type A	☑	☐ ☑	Rename column in non-empty table **A**
12	Changing type of B	☑	☑	Redefinition of view **B**
13	Addition of type(s) **E** derived from type **D**, that replaces **D** in a certain context(s)	☑	☑	Creating a new view **E** that some stored procedures will refer to instead of **D**
14	Modification of constraint on type **F**	☑	☑	Modification of database constraint **F**

This is a full and comprehensive set of modifications that require domain evolution if changes are made to the paradigm. It does not take into account modifications to the semantic domain. Proof that this is the full set of changes is found in examining the meta-metamodels used by these types of tools. Each of these 14 "atomic" types of model migration all depend upon at least one of five semantic concepts of metamodeling: the **type**, **attribute**, **association**, **inheritance**, and **constraint** (refer to Table 1). It is assumed that aspects do not contain semantic or syntax information, thus they are not included in this set of concepts. Notably, these five concepts may be used to classify the full set of first-class objects used in [83][84][85] and relational databases. See Figure 3 for the first-class object definition of GME [82].

Not coincidentally, these are all instances of the UML concepts [5] of a class diagram, and in fact, they comprise the full set of four concepts[1] used to establish syntax and static semantics in a UML class diagram. Each of the metamodeling environments of [83][84][85] use a UML-like interface when creating metamodels.

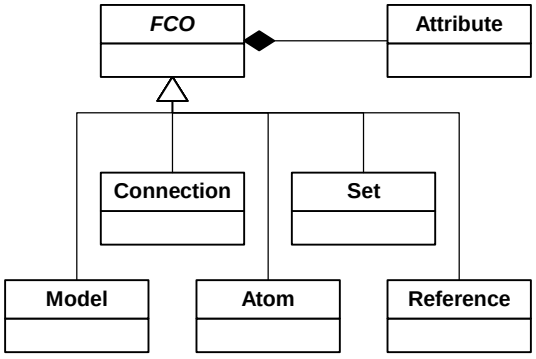


Figure 3. Meta-metamodel basic type information for GME [82]

[1] Since UML considers an attribute to be a special kind of association, there are four key concepts for static semantics, rather than the five presented in the above table. The reason for this is the different reaction of modeling tools to changes in attributes rather than changes of other types of associations.

The implementation and specific details of each type of first-class object is not important for this research. The importance of these objects is that the objects defined in the meta-metamodel determine the possible changes that are required when migrating domain models. Recall that the meta-information is stored with the domain models. Any change to the meta-information endangers the validity of the domain models, and the meta-information may be changed *only according to the meta-metamodel*. Thus the information defined by the meta-metamodel is the basis for any model migration that must take place to preserve the semantic integrity of domain models. This is an important consequence, and will be recalled when constructing the domain evolution framework in Chapter III.

Graph-Rewriting

Graph-rewriting is a useful tool for solving abstract problems that have a well-defined visualization method or that lend themselves to organization by containment or association. Graph-rewriting has been used to solve compiler theory problems [10][12], formalization of basic mathematical theory [14], not to mention graph theory and algorithm problems [15].

More recently, and to the subject of semantic mappings in high-level languages, graph-rewriting (in the form of graph transformations) has been used as a sort of universal semantic description language that allows the specification for transformation from one syntax pattern to another. This approach appeared in [16] as a semantic extension for UML. Lately, graph grammars (the syntax used to express graph transformations) have been suggested as an implementation platform for the Model Driven Architecture (MDA) [17][18].

Graph-rewriting may be cast as a generalization of string grammars or term rewriting systems. A graph-rewriting system is a set of rules that transforms one instance of a class of graphs into another instance of the same class of graphs [20]. Graph-rewriting takes two forms: replacement – where sub-graphs are replaced with sub-graphs (and consequently, the rest of the graph remains unchanged), and recreation, where every node and line in the graph are created from scratch.

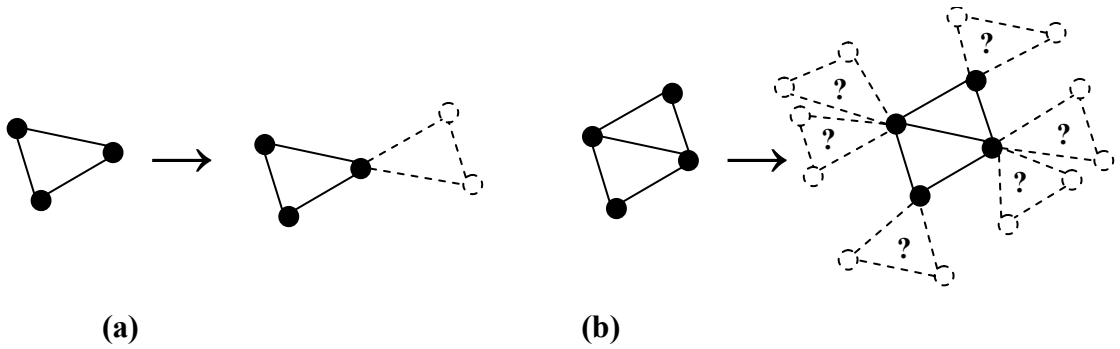


Figure 4. (a) A pattern and its transformation rule. (b) The desired transformation is sometimes difficult to specify unambiguously

Unfortunately, rules defined to carry out graph transformations, while simply drawn up, can be quite difficult to implement. A simple rule, say to replace a triangle with a bowtie can be ambiguous in its desired implementation. Figure 4(a) shows a rule that might describe the rewriting of a graph element from triangle to bowtie. However, Figure 4(b) shows the six possible outcomes of the graph as presented. Should all of these outcomes be written? Only some of them? Only ones that meet certain criteria? A graph translation language should be specific enough that such ambiguities are not present upon translation. Fortunately, there is a significant body of research in the area of graph transformation, specifically in graph transformation languages. Most of these tools

operate on a particular source graph (called the input graph) and produce an output (or target) graph. Nodes are traversed and matched based on syntax patterns entered into the graph-rewriting engine, and the output graph is produced based on the mapping between the syntax patterns and the desired output graph pattern. Well-formed arguments for the benefits of several types of graph-rewriting approaches are found in [24][96].

There are several functional graph-rewriting solutions that have been developed through years of research in graph-rewriting and graph grammars. Each language has its own syntax, model of computation, and benefits for certain problem types (e.g., selective tree rewriting, or default behavior configuration). The discussion of some mature and full-fledged languages, as well as some promising experimental languages, is given in the next few sub-sections.

PROGRES

The recognition of the ability to formulate many problems found in the realm of computer science as graphical problems led to the concept of programmed graph-rewriting systems. These systems are a set of graph-rewriting rules that are organized (programmed) to carry out some algorithm. Advances in these systems led to the development of the PROGRES language (*PROgrammed Graph-REwriting Systems*). PROGRES has been developed by Schürr and others at the University of Aachen. Much of this description is taken from [20], which provides an excellent tutorial for the use of the language; more details and interesting studies are also available from [21][22][24][26][97][98]. PROGRES is a strongly typed language whose underlying formalism is tied directly to the algorithmic graph-grammar approach [23]. The design of PROGRES was developed with the following goals,

- proper use of graphical and textual syntax where most appropriate,
- distinction between the definition and manipulation of data, as well as the use of a typed ontology to guarantee the type-correctness of graph transformations,
- reticence to require users to keep track of rewriting conflicts and backtracking status,
- independence from a programming paradigm limited to rule-oriented creation of algorithms, but rather to allow imperative programming and the specification of rule application strategies.

The underlying data model of PROGRES is similar to that of virtually all software: a directed attributed graph. A set of basic concepts makes up its metamodel, namely,

- node types – determine the static properties of the instances of this node
- intrinsic relationships – a kind of edge, or edge type, explicitly manipulated and restricted in the permitted types of its source and destination (all intrinsic relationships are binary associations)
- derived relationships – relationships determined at runtime that are defined in terms of relative associations between instances
- intrinsic attributes – the attributes of a node type, existing for all instances of the node type(s) they are associated with
- derived attributes – values determined at runtime that are defined by means of schema-independent methods on node types (e.g., the sum of attribute values, or the size of a contained set of nodes)

- inheritance – a conventional IS-A relationship between two node types; multiple inheritance is also permitted

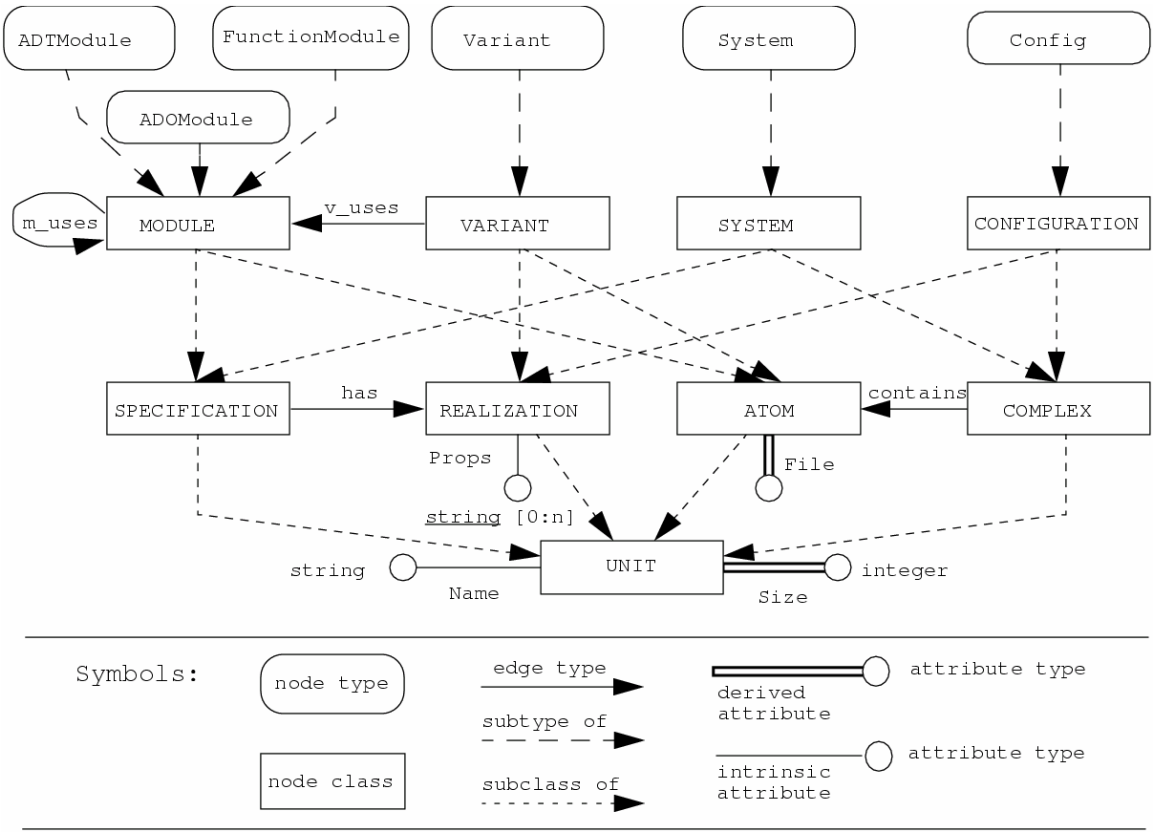


Figure 5. The meta-metamodel for PROGRES, specified in its own metamodeling language [20]

PROGRES also takes advantage of visual representation to tersely specify the semantics of symbols of its graphical notation, using square boxes, rounded boxes, and various other means to specify semantics. The meta-metamodel for PROGRES is shown in Figure 5.

As one of the design concepts for PROGRES promised, the use of visual and textual languages is divided where the language developers deemed it appropriate. Namely, functions and external types are included in supplemental textual material that is

specified at design time for the PROGRES datatypes. Example functions are provided in Figure 6.

```
from Files import

   types  file;

   functions
      size: ( file ) -> integer;

end;

function select: ( PName: string ;
                   PSet: string [0:n] ) -> string [0:n] =
   use P: string := elem( PSet ) ::
       [ propName( P ) = PName :: P | nil ]
   end
end;   (* Returns all ('PName', ?) pairs in a given property set *)
       (* 'PSet' which have 'PName' as their first component.    *)

function merge: ( PSet1, PSet2 : string [0:n] ) -> string [0:n] =
   use NSet1: string [0:n] := propName( PSet1 ) but not propName( PSet2 );
       NSet2: string [0:n] := propName( PSet2 ) but not propName( PSet1 ) ::
       select( NSet1, PSet1 ) or select( NSet2, PSet2 )
                              or ( PSet1 and PSet2 )
   end
end;    (* The result requires any property which is         *)
        (* 1) required in 'PSet1' and unspecified in 'PSet2' *)
        (* 2) required in 'PSet2' and unspecified in 'PSet1' *)
        (* 3) required in 'PSet1' and 'PSet2'.               *)

function addSize: ( Val : integer ; Atom : ATOM ) -> integer =
   Val + Atom.Size
end;
```

Figure 6. Example functions in PROGRES [20]

As evidenced from close examination of the figure, PROGRES has an extensive set of intrinsic functions useful for operations in set theory, such as "or" (union) and "and" (intersection). The functions defined using these keywords may be used on or attributed to node types in a schema. Note that all PROGRES functions are nondeterministic in their selection of individual elements when processing a set.

An integral part of the graph transformation process is the selection of appropriate subgraphs for processing. This is named a *graph query* in PROGRES, and once again teams visual diagrams with textual language for the complete specification. Queries are

textually defined similar to an interface/function definition in a textual language, with a name, parameters, and output type. The body of the query is usually a join or intersection of certain *tests*, which is the graphical portion of the specification.

```
query ConsistentConfiguration( out CName : string ) =

    (* A configuration is consistent if:
    (* 1) it contains a variant of the system's main module, *)
    (* 2) it contains a variant for any module which is      *)
    (*    needed by another included variant, and            *)
    (* 3) it does not contain variants which are not needed  *)
    (*    by needed variants.                                *)

   use LocalName: string do
      ConfigurationWithMain( out LocalName )
    & not UnresolvedImportExists( LocalName )
    & not ConfigurationWithUselessVariant( LocalName )
    & CName := LocalName
   end

end;

test ConfigurationWithMain( out CName : string )  =
```

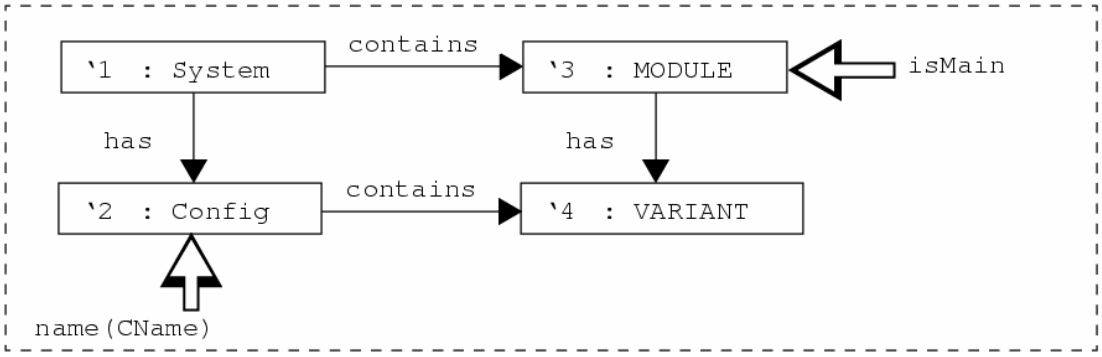


Figure 7. Example specification of a query and one test used in the query body [20]

Figure 7 is the textual definition of a query named `ConsistentConfiguration` along with one of its three tests, `ConfigurationWithMain`. This test matches any subgraph consisting of the nodes and edges bound by its identifiers `'1`, `'2`, etc. to nodes of the input graph (archetypally named *host graph* in PROGRES). In addition, constraints (called *restrictions* in PROGRES) may be placed on objects in the graph that

filter out matches that do not meet supplementary requirements (e.g., that the `MODULE` meets the `isMain` restriction).

After specifying the schema and query portions of a PROGRES graph-rewriting setup, the last step in the specification is the graph transformation. The graph-rewriting specification uses a conceptual "left" and "right" hand side, whereby objects on the right replace objects on the left. As LHS objects are denoted textually by the leading `'` (e.g., `'1` in the query previously presented) RHS objects are denoted textually by a following `'` (e.g., `2'`). Existential operators are encoded into visual shortcuts (e.g., the `MODULE` should not exist due to its corner markings). An example transformation is shown in Figure 8.

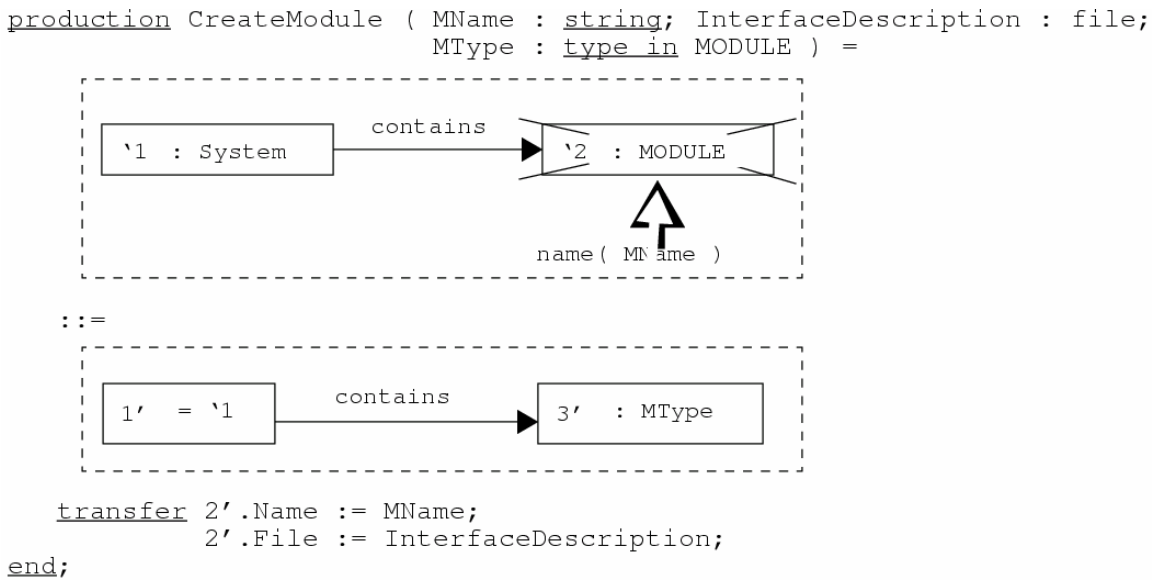


Figure 8. Example specification of a graph transformation [20]

The following rules are given for the model of computation for a graph transformation:

- all nodes in the RHS which are bound to nodes of the LHS are *preserved* without performing any not explicitly required intrinsic attribute value or edge context modifications
- all nodes and edges of the LHS which have no counterparts in the RHS will be *deleted*, including all incident edges of deleted nodes
- all nodes and edges of the RHS with no counterparts in the LHS are *added* to the host graph
- new attribute values are computed by evaluating expressions which may reference input parameters as well as old attribute values

This model of computation has several consequences. First, the output graph is not a modified version of the input graph, but is actually a graph created from scratch, where bindings from the old to the new (in an effort to preserve the nodes and edges) are required to keep each existing element of the input graph.

PROGRES is a proven language for performing graph transformations, as shown by its usage record (see [21][22][24][26][97][98]). It has a complex syntax capable of solving general purpose graph transformation problems, especially when geared at problems where graph algorithms lend themselves to the solution. Although PROGRES meets all of the design goals it set out, there are several drawbacks to the language – the primary one being its complexity. Arguably, simplistic composition and syntax relieves some of the general purpose features of the language. It does, though, discourage someone who may decide to experiment with the language. The choice of graphical constructs is also somewhat suspect as the concepts are not always intuitively obvious (e.g., the rounded corner of a node-type versus the square corners of a node-type).

Arguably, if a UML style interface (including UML metamodeling concepts) were adopted, new users might be more receptive to the language constructs.

The non-standard notation decreases the probability of sharing PROGRES transformation descriptions with someone unfamiliar with the language in order to explain an algorithm, as additional explanations would need to accompany the transformation in order to decode it. In this sense, PROGRES is not an effective domain-specific language in that domain experts cannot readily pick up the tool in order to use it in a matter of time on the order of hours. In the case of PROGRES, an intuitive interface is necessary as the tool is not the industry standard for creating simple graph transformations.

GReAT

The Graph-REwriting And Transformation language (GReAT) is a graph transformation language developed at Vanderbilt University. GReAT was developed expressly for the development of graphical language semantic translation. Much of the information for this description is taken from [28], although significant examples and design usages may be found in [18][29][30][31]. GReAT was devised as a new approach for model-to-model transformations, useful for translations from one domain to another. The following design goals were desired,

- As UML is a widely used and accepted standard for specification of classes and objects; it should use UML for specification of static structure (i.e. that data model) and integrity constraints.

- There should be support for transformations that create an entirely different graph based upon a given graph. The two graphs may have different static structure and integrity constraints.
- The new approach should be expressive enough to specify model interpreters that convert models of high-level graphical languages to low-level implementations, with no or minimal textual coding.
- The new language should have efficient implementations of its programming constructs. The implementation should have comparable efficiency to its equivalent hand written code.
- The new language should be "user friendly" and increase programmer productivity.

The language deliberately shies away from textual specifications, relying instead on an expressive graphical syntax that is related to UML in as many respects as appropriate. It also is intended to replace textual programming when applied to model transformation methods. GReAT, as a language, can be divided into three parts,

- pattern specification
- graph transformation
- control flow

GReAT uses an internal representation of a typed attributed multi-graph when describing any graph. The two basic types of objects (vertices and edges) are given basic functions that are applicable to any type, namely the `name` and `type` (for vertices), and `name`, `type`, `src` (source) and `dst` (destination) for the binary edges. The metamodel for all GReAT input graphs (and thus individual nodes and edges) is UML.

Patterns are specified using UML concepts to graphically denote the required instances and their association with other instances. Patterns can be broken down into three categories: simple, fixed cardinality, and variable cardinality.

Simple patterns are a one-to-one mapping of objects of a pattern specification to an instance in the input graph. These patterns are straightforward to specify and easy to understand. *Fixed cardinality* patterns are a sort of shorthand for simple patterns that consists of many nodes, for example matching a node with 14,000 children would hardly be feasible with a simple pattern. *Variable cardinality* patterns specify a range of possible fixed cardinality patterns to match, for example matching a node with anywhere from 3 to 10 children. Figure 9 displays examples of these three types of patterns.

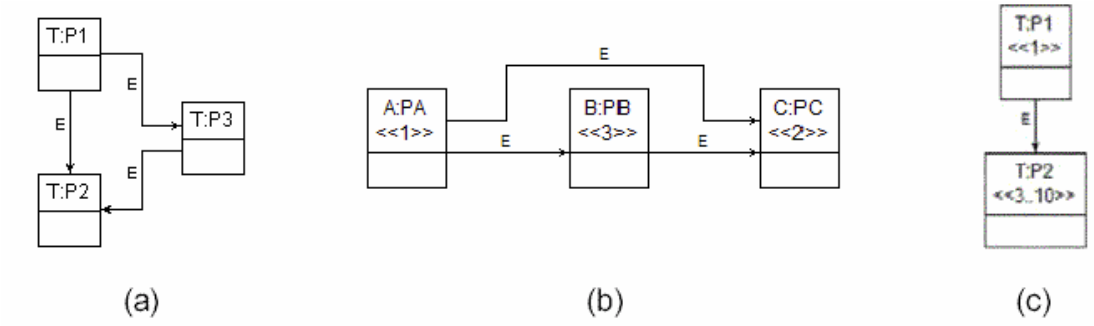


Figure 9. Examples of the three types of patterns in GReAT. (a) simple pattern, (b) fixed cardinality pattern, and (c) variable cardinality pattern

The simple approach to the syntax of pattern specifications belies a complicated semantics required to implement such constructs. For example, in a simple pattern match, what should occur when P1 contains two P2's in the pattern, but in the input graph there exists a P1 containing four P2's? The model of computation that gives the semantics of such constructs is complicated, and at times tediously specific. For brevity,

it is not explained in-depth in this document. However, it may be summed up with the following points,

- The algorithm takes as input the pattern, input graph and a partial match and returns a set of matches.
- The partial match must have at least one vertex of the pattern bound to the input graph.
- A recursive approach is applied to return a set of matches that do not include any isomorphic elements when bound to graph pattern specification objects.
- In variable cardinality patterns, the matches returned consist of a set of all matches made, none of which may be a subset of any other match.

The patterns are used as primitives in the transformation portion of the language. Graph transformations are specified by mapping from one pattern specification to another, using these three basic mapping types,

- Bind – used to match objects in the graph
- Delete – also used to match objects in the graph but after these objects are matched they are deleted from the graph
- New – used to create objects after the pattern is matched

A rule contains a pattern in the input graph and a pattern in the output graph, along with these mappings between the appropriate objects. The model of computation for the execution of the rule involves matching every pattern object marked either bind or delete. If the pattern matcher is successful in finding matches for the pattern, then for each match the pattern objects marked delete are deleted and then the objects marked new are created. Sometimes the patterns by themselves are not enough to specify the exact

graph parts to match and we need other, non-structural constraints on the pattern. An example for such a constraint is: “an attribute value of a particular vertex should be within limits.” These constraints are known as guards, and are described using Object Constraint Language (OCL) [6] as it is a widely used standard and is directly related to UML. There is also a need to provide values to attributes of newly created objects and/or modify attributes of existing object, called “attribute mapping”. Attribute mappings are a set of assignment statements made in a procedural language that uses C syntax and provides interfaces to the bound objects through their names on the diagram.

The third and final division of GReAT is its control flow. Based on patterns matched (or not matched) in a rule, the flow of control can change from one possible rule to another. This allows for conditional processing of input graphs. In order to increase the efficiency of the execution of a graph transformation, bindings may be passed from one rule to another in order to lessen the search space on the graph. There are also rules for the model of computation of the execution of sequenced rules. With GReAT it is possible to model both deterministic and non-deterministic behavior of an execution. This can be important in translations where the outcome of the algorithm is different based on the order of application. Some algorithms require ordering of rules, while others do not, and thus deterministic behavior is modeled at the discretion of the modeler [28].

The underlying semantic domain of GReAT requires the use of the UDM [32] package (also distributed by Vanderbilt University). UDM, the Universal Data Model, is a generic programming interface to data created according to metamodels. UDM generates a domain-specific API, depending on the metamodel, which is a wrapper for

the generic API that all UDM objects may be accessed with. In addition, UDM objects may be stored in different formats of persistency, for example XML, or GME formats.

While GReAT effectively uses the UDM execution model and API to perform its transformation of the graph transformations the input and output graphs are required to be in UDM format. UDM is beneficial in that reads XML files of a certain DTD (which UDM generates from the metamodel of the input/output graph), so the production of graphs in this format is not as difficult as a proprietary format. However, it can still be a barrier to the introduction of new users to the project. An overall illustration of the GReAT framework is given in Figure 10.

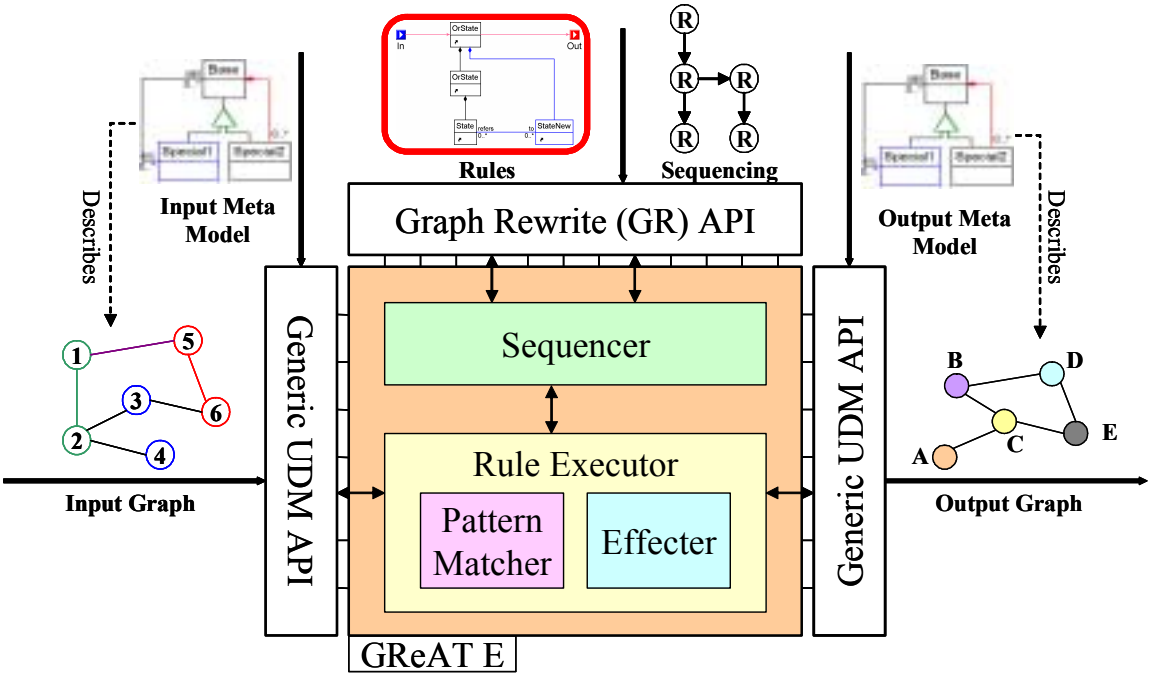


Figure 10. The GReAT framework

Overall, GReAT is a powerful rewriting concept. However, there are some drawbacks. The first is that there is no stable, proven release, and it is still undergoing

development and research. Second is the lack of an isomorphism operator that may be used as a wildcard specification for direct copying of matched objects. A final drawback is the runtime environment, which requires the underlying UDM to be installed and attached to the GReAT framework. However, this final penalty is not significant when compared with the others.

BOTL

The Bidirectional Object Transformation Language (BOTL) is a graph transformation language developed at the Technical University of Munich [25]. BOTL is designed to be a general purpose graph transformation language, but also has leanings to the model driven architecture (MDA) proposed by the OMG. In addition, BOTL sets out to achieve bijectiveness (or to at least allow for the possibility of bijectivity) for any transform created using its framework. Three properties of a graph transform language BOTL designers most desire to implement are

- Applicability – the property that the application of a rule set for a given metamodel doesn't cause any conflicts for any arbitrary source model,
- Metamodel conformance – the application of a rule set should not generate output models that do not conform to the proper metamodel,
- Bijectiveness of transformations – the possibility that the rule set can be inverted and applied to an output set and thus produce an isomorphic version of the original input graph.

BOTL uses as its underlying storage model a UML style interface. This allows metamodels that are defined using UML to be used by the BOTL framework (after some

slight modifications). Patterns are specified using UML style diagrams that intuitively provide the meaning of the pattern to a graph transformation domain expert.

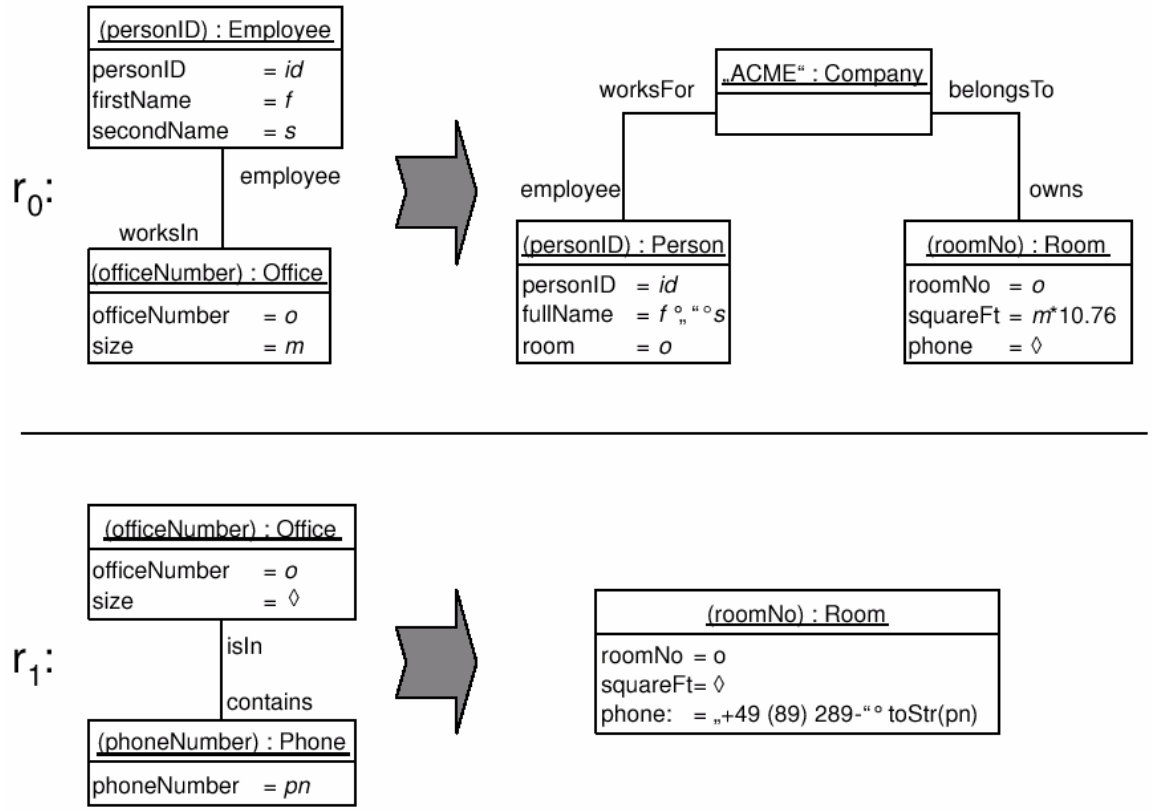


Figure 11. An example BOTL rule set, $r = (r_0, r_1)$ [25]

As evidenced by Figure 11 BOTL rules are composed of a LHS and a RHS. Moreover, the sequence of a set of rules is preserved during the execution of a BOTL transformation. That is, r_0 will be executed before r_1.

The process used by BOTL to return subgraphs is similar to that of GReAT. The patterns are searched across the entire input graph, and a set of matches is returned for use within the rule. This is shown by Figure 12, and the result is termed a *model fragment match*. Model fragment matches are used for obtaining subgraphs of the

original input graph for transformation as well as the creation of subgraphs in the output graph.

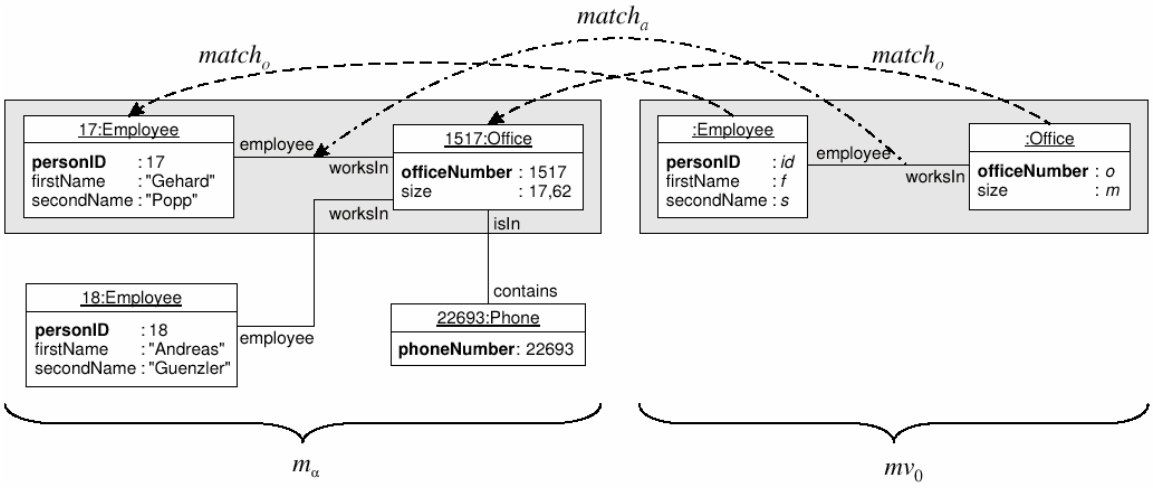


Figure 12. A model fragment match [25]

The problem of mapping attributes (and creating default attributes) is attacked with BOTL-provided concatenation and string operation methods that are by default given in the framework. However, the architects do provide the caveat that the concatenation and string operations are not fully capable of all bijective operations. For instance, they cite the possibility that the output graph may require attribute values to be concatenated, or numbers summed to produce final values. There is no easy way to take these strings and break them down into their original values, however.

All output model fragment matches are merged when the output model is created, which allows the underlying model of computation to consider each individual model fragment match as its own subgraph, and pass along the full set of created subgraphs for intelligent insertion into the output graph after all rules have been executed. Proofs for the feasibility of this mode of implementation are provided in detail in [25]. In addition

to these basic concepts of sequenced rules, model fragment matches, and attribute manipulation BOTL provides advanced features such as variable introduction and representation for more complex transformations.

Given the basic and advanced BOTL formalisms, the tenets of applicability, metamodel conformance, and bijectivity are discussed in detail in [25]. No further explanation is given in this document of these concepts, but some observations are made. The metamodel conformance of output graphs is not guaranteed without in-depth analysis of the transformation. This is technically infeasible, due to the complexities of the output metamodel (including its interdependent constraints and contextual syntax rules) and the unknown nature of the set of input models. This observation is also made in [25], but steps are provided to aid in elementary conformance guarantee checks.

Overall, BOTL is an extremely well-defined graph transformation language, aimed at MDA style approaches to graph transformations (between two well-defined metamodels). The benefits of BOTL revolve around the easy to read rules and patterns, as well as the simple design for the ordering or rules, and in general the readability of BOTL specifications. The failings and drawbacks of BOTL are few, except that it too is emerging research, and neither proven to work, nor widely used. In fact, no public releases of BOTL exist at the time of this writing.

XSL

The eXtensible Stylesheet Language (XSL) is a scripting language designed for the translation of XML data to other formats and maintained by the World-Wide Web Consortium (w3c) [93]. One of the most common uses of XSL is to transform XML formatted data into an equivalent HTML format (e.g., turn an XML database into an

HTML table with visual effects to accentuate or hide certain attributes). XSL also includes language constructs that provide an interface useful when producing XML documents from other XML documents. These language constructs abstract away the creation of properly formatted XML elements, allowing for focus on the values prescribed by the XSL Transform (XSLT).

XSL is a Turing complete language [34] that is implemented as a functional language rather than a procedural language. As suggested by [94] XSL is such a general purpose language that it can be described as a general tool for transforming the structure of an XML document (therefore unrestricting the style of output). XSL has created a significant amount of excitement since its introduction in 1999, and with good reason. Two compelling features (which were design concepts) of XSL are,

- Separation of data from presentation – the separation of concerns is an ever growing paradigm for the development of software. XSL provides separation of data from its visualization, which allows for data reuse in multiple applications (e.g., using a database of cities in a map-drawing application as well as weather forecasting)
- Transmission of data between applications – data interchange has heretofore been limited to two possibilities: formatted ASCII or proprietary binary formats. With XSL it becomes possible to reformat data created by one tool to data usable by another tool

XML is a suitable input graph format for discussion in this dissertation for several reasons. The inherent *n-ary* tree structure of an XML document allows an XML document to be properly classified as a graph. Also the widespread use and easy-to-learn

nature of XML make it likely that input and output model database will have an XML representation, and an XML schema that can be modeled as a metamodel.

There are three portions to the execution of any XSL transformation, the *input*, *output*, and *stylesheet*. The input is an instance XML tree, and the output is of an unrestricted metamodel (although for the purposes of this discussion we will use an output metamodel of XML format also). The stylesheet is the meat of the transformation process.

Technically, a stylesheet is a functional program written using the XSLT language. The difference between XSL and XSLT is subtle, and not necessary to understand for the purposes of this dissertation. Therefore "XSL", "XSL transform" or "XSLT" will be used interchangeably. An XSL transform is a program written in a language that is itself in XML format. The fact that an XSL transform is itself an XML document is appealing because XML generation APIs can be used to create the XSL document, rather than a textual method.

XSL *stylesheets* take an input XML document and use nodes that are matched from that document to create output nodes. The stylesheet is executed by an XSL *processor*. The processor selects the root node for processing, and then recursively selects each of the child nodes of the root. Processing takes place, then, after each node is selected by the processor and before other nodes are selected by the main execution. The nodes, once selected, are given to the body of a *template* for further processing.

Nodes are selected for processing in a template through XPath expressions [95]. The template takes a set of nodes and performs further operations on it, including calling other templates, production of the output graph, or ignoring elements contained in the

current set of nodes. A stylesheet may have any number of templates, and the XSL processor determines the order in which those templates are applied through a formula that compares the complexity of matches and attempts to apply the most complicated first. Each node selected by the processor is compared against each possible template, and as soon as a match is found no further templates are applied. In this way, the most complicated matches are attempted first, and the least complicated are reserved for the end of the processing. Also, nodes are processed only once by the XSL processor, unless explicitly selected and called by another template. This is designed to prevent multiple matches, but allows for recursive processing, if so desired. Note that when recursion occurs, it is on the input document, not on any generated output.

Matching criteria are created using a terse selection language named XPath [95] that was developed alongside XSL. XPath is similar to a graph matching language in its ability to select nodes for processing based on,

- Relative context
- Attribute value
- Type

An additional feature of the XPath language is that it allows nodes, once selected based on these three kinds of values, to be further filtered before passed on to the stylesheet for processing. An example is given below,

```
select="model[@kind='Camera']"
```

Figure 13. Example XPath expression inside an XSL select statement

All XPath statements operate on the currently selected node. The XPath statement given in Figure 13 selects all child elements of the current node that is of type `model`. That set of elements is then filtered using the statement found in `[]`, and only those elements in the set who have an attribute (`@`) of type `kind` with value `Camera` are returned to the processor. XPath provides a syntax that allows the traversal of trees through containment, as well as simultaneous filtering on multiple levels of that containment. Set operations are also possible, as sets can be joined together (union actions), and filters can be placed anywhere in an XPath statement (intersection actions).

In addition to context matching, XPath also provides a body of functions that are useful during selection of nodes. These functions can help in the guarantee of uniqueness, selection of nodes not in this context hierarchy based on unique-id, string parsing utilities, and rudimentary mathematical operations. Location and selection of unique elements allows XPath to traverse multi-graphs that are encoded in XML format. XPath functions (like expressions) are composable, allowing for arbitrarily long expressions and functions to be combined to select a certain set of objects.

Overall XSL is an extremely powerful language capable of any calculation. Its terse syntax and composable nature make it ideal for code generation. Although difficult to understand for programmers educated primarily in procedural programming, its acceptance as a language in the XML community has built a significant user base. Also, the language is fairly easy to read for a beginner, albeit much more difficult to write without an extensive manual or example code. Although XML is capable of the traversal of tree-encoded multigraphs it is not as efficient as its traversal of simple tree graphs due

to the need to search the entire input XML document each time an association is traversed.

Table 3. Taxonomy of reviewed graph-rewriting languages

Feature	PROGRES	GReAT	BOTL	XSL
UML Interface	☐	☑	☑	☐
Graphical Component	☑	☑	☑	☐
Textual Component	☑	☑	☑	☑
Attribute Management	☑	☑	☑	☑
Can execute with compile-time engine	☐	☐	☐	☑
Runtime engine only	☑	☑	☑	☐
Requires special domain model format	☑	☑	☑	☑
Allows wildcard matches	☐	☐	☐	☑

Table 3 is a taxonomy of all the graph-rewriting languages reviewed in this paper. Each of these languages is full-featured, but – like metamodeling languages – each has its own benefits and drawbacks. For example, XSL is not suitable when requiring a domain expert to implement a model migration solution if she is unfamiliar with XSL, as it is not an easy language to learn, as well as not graphical in nature (and thus not integrating well with a graphical metamodeling environment). However, if the graph-rewriting language was generated from another graphical environment, it would be easier to generate and run XSL code than the others because it is a textual language only, and can execute with an

engine linked in at compile time. The important lesson is that the appropriate graph-rewriting language should be chosen for the appropriate problem.

State of the Research

The final result of any model migration tool will be domain models that are semantically and syntactically correct in the new domain. This chapter examines the state of research for another type of domain evolution: database schema evolution. A significant amount of research has been performed on database schema evolution, and these findings, as well as the history of the research, are documented here. There has also been research on metamodel based graph transforms which is discussed in the last portion of the section.

Database Schema Evolution

As previously mentioned, the model of a database schema is roughly equivalent to what MIC researchers call a metamodel. The main difference between an MIC metamodel and a database schema is that database schemas are a domain-specific modeling language for database development. However, they serve as a descriptor of metadata, and are therefore subject to the same difficulties when that metadata changes. A significant body of research exists that describes frameworks and methodologies for transforming data to work in different evolutions of a schema.

Schema Evolution Defined

The problem of schema evolution was first encountered in the realm of traditional database systems [35]. Solutions to this problem emerged as data transformation

languages [36][37][38], which operated on the data as a high level language. The database manager could migrate the data to the new schema, but was required to perform a great deal of work in the translation of the data integrity requirements into the high level language.

Note that schema evolution, while related to schema integration, remains a slightly different problem. For one, schema evolution is concerned with retaining old data for use with a new schema. Schema integration, however, is concerned with integrating multiple databases for use with multiple usage points (i.e., clients), and ensuring usage across changes to the number of databases that integrate, along with their individual schemas (usually resulting in a global schema with namespace support of some sort). For more information on schema integration, see [39].

Schema-Based Evolutions Emerge

In order to alleviate the difficulties in translating the data integrity requirements into the transform language, solutions to the schema migration problem emerged not as transformation languages, but instead as databases with internal structures that were tolerant of the evolution of schema. Examples of some of these databases were ORION [40][41] and GemStone [42]. Although these databases claimed to solve the problem of schema evolution, their claim must be rejected for some basic reasons.

First, although the built-in transformation engine supported all schema changes that were made in the database, the list of possible schema changes that could be made was limited to only a few primitive items. This meant that things such as complex type changes were not supported. Second, when several primitive changes were made sequentially, data was not able to be preserved between them. Changes such as data type

replacement meant that the data had to be deleted first, and then the new type created – resulting in the loss of all of the previously deleted data. The way this problem was solved was with custom code that could preserve the data for later integration with the database. Thus, while many of the problems encountered by early databases were solved by ORION and GemStone, the basic problem of translating the data migration into customized code still remained as a possibility.

Later, more mature databases were developed that provided better solutions than ORION and GemStone. By providing a database environment that supported multiple versions of schema, data was able to exist according to several different schema at the same time. Examples of these databases are found in [43][44][45][46][47][48][49][50]. While this did provide the ability to preserve the database data intact through the evolution of the schema, it unfortunately decreased performance of the database, and had large overhead requirements for space, in order that the retrieval and storage algorithms could decide between schema versions. Other databases preserved one data storage copy, but simulated previous schema interaction versions by providing table views that corresponded to the old schema [51][52][53]. This enhanced performance, but strictly limited changes that could be made to the schema (e.g., primary keys could not be changed). Also, each point of interaction with the database required its own view, which created headaches for view administration between users. In an attempt to improve performance (and decrease downtime between schema changes) [54] describes a method for dynamic schema evolution.

Finally, coming full circle to the originally defined translation languages, the TransformGen system emerged [54][55]. Designed for use by Gandalf programming

environments [56][57], TransformGen supports the evolution of abstract syntax grammars. Analogous to type definitions (i.e., schemas), abstract syntax grammar changes that were supported by TransformGen are analogous to those supported by ORION and GemStone. However, one key difference remained: TransformGen also allowed for manipulation of the generated transformation code. This meant that schema experts could modify the transformers to have the exactly desired effect on the data. This was taken advantage of in [58] where TransformGen was modified to support flexible transformation of databases (rather than the abstract syntax grammars).

Most recently, such approaches as [59][60][61] attempt to utilize representations of the schema as the language for the translation (Entity-Relationship diagrams, for example). This direction is promising (from the perspective of a domain-specific environment for migrating domain-specific data), but not yet mature. Also, only a small amount of research has been committed to maintaining the semantic correctness of the database constraints [62]. Thus far, the concern for semantics in database schema migration has been limited to the translation of data, and little thought has been given to the translation or maintenance of constraints.

The research in database schema migration was useful to the database community, but does not translate well to formal modeling. One reason for this is the lack of visual metamodeling in the schema definition of most tools and languages used to migrate the databases. Second, the speed with which SQL queries can be performed allow for the ability to select large bodies of data and place them into the new database in the proper format. Formal modeling lacks a general purpose data selection language (that translates across all modeling environments) and is heavily dependent on the metamodels for

definition, ruling out database schema migration solutions as extendable to implement model migration solutions. Rather, the lessons learned in database schema migration should be applied, namely that the matching of abstract syntax grammars, and the mappings between schemas, are the preferred method of migration.

Ad hoc MM Solutions

These solutions are not engineered, but rather evolve out of necessity. *Ad hoc* solutions are often devised through low-level programming for a fixed domain change. The important note is that all of these processes are defined *for a certain schema change*. That is, no general-purpose tool for modeling the migration of customized modeling domains once those domains change. Solutions exist for two types of problems: between two well established domains, and instances of one domain evolution.

MM For Well-Established Domains

Certain tools exist that can perform data migration between similar domain-specific environments. Examples of such tools are Microsoft Word and Excel, which can convert data stored in Microsoft's proprietary format to such formats as those for Borland, or perhaps to HTML format for web display.

Another tool example is more sophisticated, because it is a standalone program that converts model repositories into new formats, as opposed to Word's ability to convert itself into another format. This tool, called MetaIntegration Works [63] allows a user to query databases of models, and produce as output a database of models semantically transformed into a new schema. Examples of these transforms are from Rational Rose [64] to Oracle Designer [65].

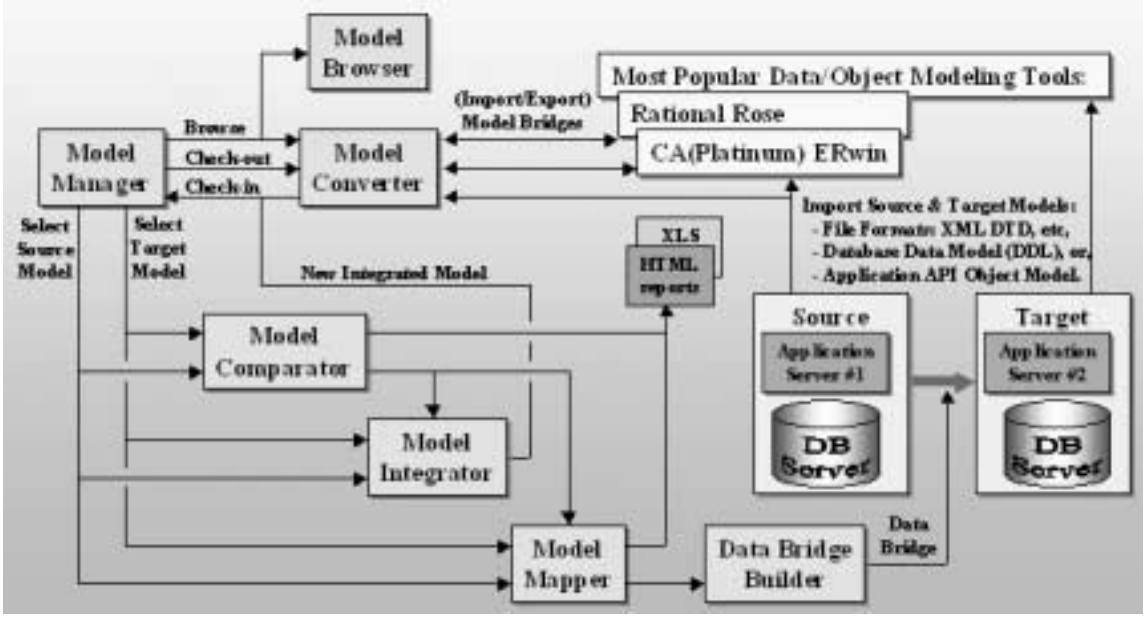


Figure 14. Functional overview of MetaIntegration Works [63]

However, these migration solutions are based on well-defined meta-models. Figure 14 displays the overview of the functionality for MetaIntegration Works. Note that the source and target databases are limited to an *a priori* list of modeling tools. Any attempt to use models defined under any customized paradigm in conjunction with the MetaIntegration software would be unsuccessful. In order to use the software, MetaIntegration would need to examine the customized paradigm semantically, and write new components to and from each possible modeling tool.

In addition to their dependence on existing metamodels, these *ad hoc* migration solutions do not provide the ability to customize the translation that takes place. For example, a concept that exists in Microsoft Excel, but not in Borland Quattro-Pro, is transformed with a loss when encountered in data. The definition of this transform is not available to the user, because it is locked away in a binary file.

MM For Instances of One Domain Evolution

The other, and far more abundant, MM solution that already exists is the evolution of a particular domain. Since CBSs require system data to operate, upgrades to CBSs require either MM, or rebuilding the system data by hand. Most CBS upgrades, however, are not demanding enough to require a general solution to MM. In fact, a solution is usually found for this particular CBS evolution, and then carried out upon upgrade.

An example of this is a database system upgrade. A flat-file inventory database exists on a VMS machine, and the CBS is being upgraded to run on a relational object database such as MS-SQL Server. Each file of the flat-file database maps to a relational table, but there is no simple routine to import unknown flat-file formats to SQL Server. The usual solution is to transform the flat-files into some format that SQL Server can recognize, and then import the files.

This solution is typically written out as numbered directions followed by a trained operator. Consider, for example, the following customized schema migration utility (shown in Figure 15) that transforms the contents of a Netscape Directory Server from version 4.1 to 4.x after the Database and Server have been upgraded [66].

Now, in several key areas, this is a good model migration solution. The solution is custom designed for the evolution, and the instructions are aimed at a Netscape Directory domain user (not a schema-migration domain user). However, it does fall short in one major way: the solution is not created through modeling the meta-data and generating a migration script/executable. A better solution here is to model the added

schema elements, and automatically create the standard object classes described in section 3.c.iii of the non-standard schema operations.

```
Upgrading standard schema:
   1. If you have not already done so, complete "Step 1: Back Up Your
      Configuration and Database" and "Step 2: Upgrade the Server".
   2. Copy all the files in the following directory:
      <NSHOME>/bin/slapd/install/config/
        except slapd.user_oc.conf and slapd.user_at.conf, to the
        directory where your 4.0 schema files are stored. By default
        this directory is:
      <NSHOME>/slapd-<ServerID>/config/
   3. Restart the Directory Server

Upgrading non-standard schema:
   1. If you have not already done so, complete "Step 1: Back Up Your
      Configuration and Database" and "Step 2: Upgrade the Server".
      Make sure you create a backup copy of your schema files.
   2. Copy all the files in the following directory:
      <NSHOME>/bin/slapd/install/config/
        except slapd.user_oc.conf and slapd.user_at.conf, to the
        directory where your old 4.x schema files are stored. By
        default this directory is:
      <NSHOME>/slapd-<ServerID>/config/
   3. Examine the copy you made during "Step 1: Back Up Your
      Configuration and Database" of the old 4.x slapd.at.conf and
      slapd.oc.conf files to discover all the schema additions that
      you made.
         a. If you added attributes to slapd.at.conf, you need to add
            them to the new Directory Server using the Directory
            Server Console.
         b. If you added new object classes to slapd.oc.conf, you
            need to add them to the new Directory Server using the
            Directory Server Console.
         c. If you added attributes to standard object classes in
            slapd.oc.conf, then you must do the following:
               i.  Using the Directory Server Console, create a new
                   object class that allows your custom attributes.
              ii.  Place this new object class on every entry in your
                   directory that uses the custom attributes.
             iii.  See the Netscape Directory Server Administrator's
                   Guide for information on adding object classes and
                   attributes.
   4. Restart the Directory Server.
```

Figure 15. Instructions for ad hoc model migration during system upgrade for Netscape Directory Server [66]

Another major problem is the possibility of human error. If even one error is made during the migration process, the entire process must be repeated from the copying of the old schema files. In addition to being laborious to perform the steps, it is also

frustrating for a domain user to find out that the migration was not successful after she thought she had done everything correctly. Modeling the system and generating the migration operations could help to make the migration process smoother.

Universal Language and Interchange Formats

As standards emerge through organizations such as the International Standards Organization (ISO) and the World-Wide Web Consortium (w3c), some discussion must be given on the thought of a universal language that could be used for exchange of data and the expression of semantics for a particular domain.

Compilers

When high-level computer languages emerged as a plausible programming method, a formidable problem was the production of compiler optimizations that translated the high-level commands into low-level (i.e., machine code) commands [67][10]. These optimizations were performed *after* the language performed any high-level optimizations (such as removing dead code) and optimize such machine-specific usages as register allocation.

As languages and architectures proliferated in the early years of computing, the creation of compilers and optimizations for different architectures required more and more work [11]. This was not because languages became more complex, nor because the architectures became more difficult to understand: the problem was that the amount of work increased in an n x m fashion. In other words, for the m^{th} architecture, n different optimizations had to be developed, one for each language that might be used with that architecture.

Computer programmers realized that this problem was inconvenient, if for no other reason, because it was confusing whose responsibility it was to produce optimization characterizations: the architecture, or the language. The solution that eventually emerged was the creation of a universal grammar for a particular architecture. Using this grammar, it was possible to *generate* the compiler optimization with an architecture independent program.

The compiler optimizer took in as its input not the computer code (in textual format) but instead took the *semantics* set forth by that code, as represented by a universal semantic graph. This semantic representation was developed independent of architectures, with languages and algorithms kept in mind [12].

The optimization generator and semantic representation allowed for the reduction of an *n* x *m* expansion to an *m*-hard problem, placing the burden of "expansion" of languages to the architecture developers, and leaving the language developers "in the clear." Incidentally, these optimizers used graph-rewriting and pattern matching as their method for generating the final code [11][70][70].

Using a similar approach to solve model migration is tempting, but overall infeasible due to differences in the semantic domains. First the universal semantic graph defined for the compiler optimization problem was extremely low-level, as it was dealing with the optimization of machine code. This was beneficial, in a way, as a limited set of operations could be used, and the object code out of the compiler was on about the same level of abstraction as the machine code. However, the domain specific nature of MIC environments tend to attach complicated semantics to a few domain concepts. Translating these complicated semantics into a sort of universal semantic graph would be

extremely time-consuming, and would still involve the mapping of that semantic graph back to the particular MIC environment. Also, translation between domains does not always preserve semantics, and thus there would need to be different inputs from the semantic graph. An approach to input/output to a common structure can be beneficial, however, between very similar domains, as is exemplified by the next section dealing with CDIF.

The CASE Data Interchange Format (CDIF)

The explosion in the number of Computer-Aided Software Engineering (CASE) tools in the 1990's led to a similar problem to that of MM: the interchange of models between modeling tools. Dozens of tools could generate skeleton code and documentation from class diagrams and other UML-based software engineering notations. Of course, once a tool was chosen for a particular application, then that tool had to be used for the duration of the application's life. This was because the UML-based notation was stored in proprietary formats that were not easily interchanged between tools.

Software engineers realized that it was not feasible to create a solution from each individual CASE tool to every other tool, if for no other reason, the enormity of the task (recall the expansion of the number of solutions to the compiler optimization problem as described in the previous section). Another, more important, reason is that it would not be possible for CASE tools written in 1994 to provide functionality to export models to those written in 2004. What was decided instead was to develop a standard interchange format between CASE tools [71]. It would be the responsibility of each tool to provide the ability to import models in this format, and also to export their own models to this

format [71][72]. Incidentally, CDIF used a 4-layer architecture similar to that implemented by UML, and thus had metamodels, and a meta-metamodel, etc., so the data was well structured, and could be examined for well-formedness.

This is a viable solution to MM between tools that are quite similar, and that are modeling environments of the same (or very similar) domains. An example is the tools that deal with the modeling and simulation of hybrid systems (e.g., HyVisual [73], Charon [74], Checkmate [75]), which commonly export and import using the Hybrid Systems Interchange Format (HSIF) [76]. In this way it becomes much simpler to encode domain models in this universal (to the domain) semantic graph, because the concepts are on the same level of abstraction.

However, to use such an intermediate language to attempt to program *all* possible domain models is simply not feasible, due to the abstractions required, and the difficulty to get everyone who programs anything in the world to use the common language.

Model Transformations And MM

Despite the dearth of MM solutions that are model-based and paradigm independent, a significant body of research exists which is aimed at solving a problem quite similar to MM, which is model transformation. This research uses the ideas of graph matching and graph transformations, and also integrates the idea of UML as a basis for the description of these graph transformations. The research exists in the form of theory and proofs of concept rather than as packaged toolsets, so the division of concepts is denoted by researcher rather than toolset name.

Lemesle

In [77], Lemesle describes some basic formalisms for the development and existence of transformations rules that are based on metamodels. The formalism, which is termed sNet in the paper, is based on the notion of a semantic graph network wherein only nodes and edges of the semantic graph can be manipulated. Using metamodeling techniques these nodes and edges are mapped onto the concepts and relationships of a domain. The notation for sNet is given in Figure 16.

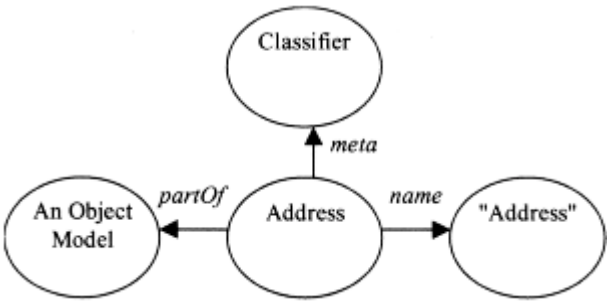


Figure 16. sNet notation [77]

There are four entities in this figure,

- the Address node, which is the representation of an address class,
- the "Address" node which is the name of the class,
- the Classifier node which is the type (or meta) class, and
- the An Object Model class which describes the context of the Address class (i.e., in what context the Address class is applicable, or may be found).

Not based on UML standards, this is a tedious, albeit sufficient, method in which to specify the existence of graphs that are abstractions of domain concepts. Other methods of specifying these objects are provided in the paper, some of which are easier to

read and explain, but overall they are translated into some form similar to the above (although possibly in textual form) for processing with the transformation engine.

Using the *meta* nodes of the sNet formalism, transformations are textually specified using a syntax similar to that of first order calculus. An example transformation would be,

```
Classifier(c) name(c,n) -> Table(t) name(t,n);
```

Figure 17. An example transformation of a graph specified using the sNet formalism [77]

This transformation specifies that for every node of type *classifier*, a *table* type node should replace it, and that the name of the previous *classifier* should become the name of the newly created *table*.

This work is an early example of the need to link the transformation of typed-graphs with the metamodels that give their types. However, the transformations are not graphically specified (although research in this area was listed as ongoing), and the graphs and their metamodels are not created using a UML like notation.

Milicev

In [78] and more briefly in [79], a UML-based technique is described that transforms models in order to take advantage of existing model interpreters. The advantage to this technique is the application of one model interpretation (i.e., a particular dynamic semantics) across similar domains can be achieved by changing the domain models rather than the model interpreter. As more and more interpreters are available for

these domains, the advantage to performing a model transformation as opposed to maintaining several different versions of many interpreters becomes apparent.

This technique relies heavily on the UML specification of metamodels. Given two metamodels (i.e., domains) implicitly related, a series of UML-like object diagrams may be used to specify a mapping between them. These diagrams use an extension of UML, so as to allow for all of the UML class diagram features (such as containment, inheritance, etc.) to be used in describing the traversal of the domain models according to the extended classes (e.g., a ForEach class describes a loop of all instances of a certain type of class).

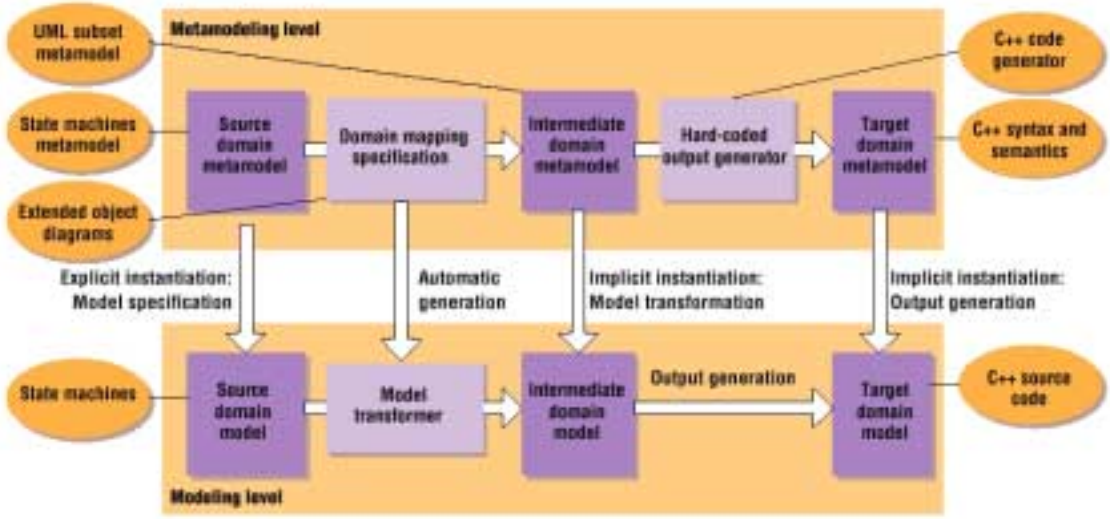


Figure 18. Example transformation overview using Extended UML Object Diagrams [57][58]

Figure 18 gives a graphical representation of the transformation of domain models from one paradigm to another (in this case, both are C++ programs)(Figure 18 describes a specific example transform, and not transformations in general). Figure 18 deals with source and destination metamodels, and their corresponding domain models.

However, the real "model migration" in Figure 18 (and consequently, [78][79]) takes place at the intermediate domain level. Here, the idea is to transform all domain models of the source domain into some intermediate domain, and to interpret that domain to the final target domain (which is C++ source code).

In this research many worthy research conjectures are shown to be effective, such as

- the use of multiple transformation specifications which introduce the opportunity for code reuse within a transform, and
- specialized transformation models that indicate iterative processing (ForEach, ForAll, etc.).

The research outlines some possibilities for generality in the specification of model migration, but does not follow through on those concepts. For example, the solution's "destination" domain model never changes, but is essentially a modeled C++ program. Also, no mention of the semantic translation is made. The semantic meaning of the source domain models is not considered as a factor in the production or generation of the model transformation. Therefore, the difference between it and any interpreters of the universal intermediate domain (there is only one listed in the figure) would also not be considered in the solution. Due to this omission from the solution, the modeler would require intimate knowledge of the source domain (i.e., the modeler would in effect be the semantic interpreter of the source domain).

This method – while acceptable for the previous transforms – exists as a particular instantiation of a framework that is proposed in this dissertation, and not as a general-purpose extensible solution. Also, its reliance on an intermediate domain does not

translate well to a general purpose solution, as described in a previous section discussing the drawbacks of a universal language for model interchange.

Thomasson

In [80] a transformation from Simulink/Stateflow to CSL is presented, and the data translation process is modeled using custom-defined transformation rules. Although the rules were not generated from the models, a comparison of the rules with the pseudocode they describe relates several necessary conditions for a successful translation between domains, viz,

- management of attributes,
- ability to query textual information and perform transformations on it,
- well-defined structures for creating and referring to new classes, and
- conditional creation and assignment.

The lessons learned in this transformation approach are that textual information is an important portion of the specification of the transformation – especially if the source or destination metamodel defines a textual rather than graphical language.

CHAPTER III

A DOMAIN EVOLUTION FRAMEWORK

The key to managing domain evolution is found in understanding how domain models are created. All software is created in some language; domain models (i.e., formally defined domain-specific software) are created using a domain-specific modeling language [2]. This domain-specific language is also specified using some language, and that language itself is specified using some language. It is easy to get confused about which language we are talking about at which time, so at this point we will define the role of each language, and how it is related to the formally defined domain models and their evolution.

The evolution of domain models is required when changes are made to the paradigm in which those domain models were created, i.e., when the domain-specific modeling language changes fundamentally in its ontology, syntax or semantic intent. Immediately, the intent of the domain expert who created the domain models becomes questionable: once migrated to the new language, will (or should) the intent remain the same, or will (or should) it be changed? It is even possible that the existing intent should be preserved, but using the new syntax to preserve the originally intended semantics in the new model of computation.

Figure 19 gives a graphical description of this dilemma. As the figure shows, there is some difference, Δ_1, between the two domain-specific modeling languages (else, no domain evolution would be required). The size of this difference is the metric as to whether or not the translation of the domain models from one domain to the next is

termed model migration or model transformation. There is no hard definition of the difference between model migration and model transformation, but when taken to extremes there are definitely *evolutionary* changes, where the Δ_1 of the two domain-specific modeling languages is perhaps only one keyword, and *revolutionary* changes, where the paradigm ontologies have few (if any) keywords in common, and the two syntaxes are more often unrelated than similar.

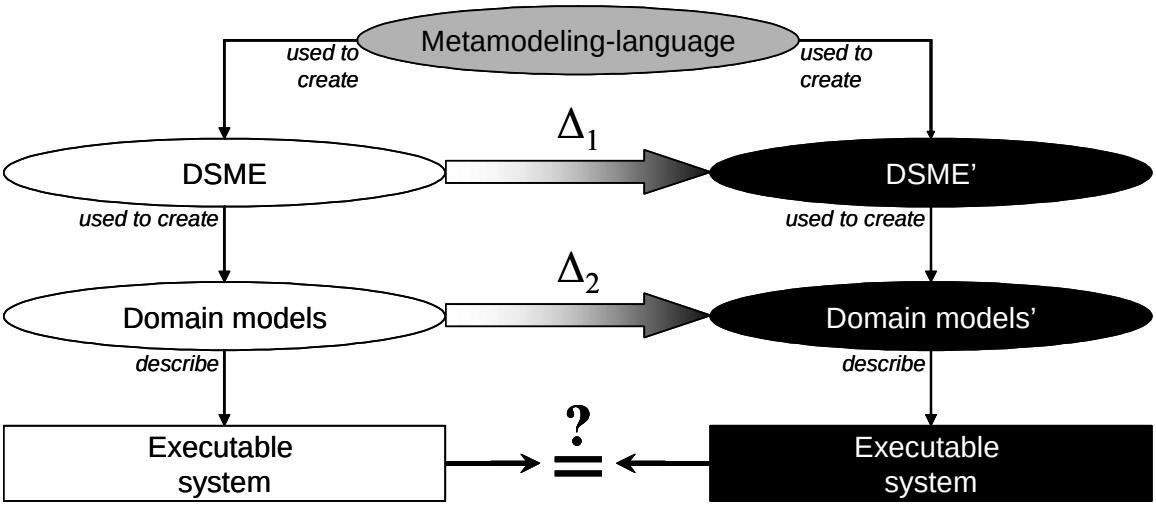


Figure 19. The DSME evolution specification (Δ_1) and the generated model migration executable (Δ_2)

In Figure 19 the two thick arrows in the center of the diagram show the evolution of the DSME and the domain models. Regardless of the reason for the evolution of the DSME (as described in Chapter II) the difference between the DSME and the DSME' requires that the domain models be updated. The migration of all the domain models completes the domain evolution process.

The objective of the domain evolution framework is to formally represent the DSME evolution as a *specification* (as denoted by Δ_1) and generate the model migration *executable* (as denoted by Δ_2). This point bears emphasis, as there is a crucial difference between Δ_1 and Δ_2. The evolution specification, Δ_1, is a formal representation of the difference(s) between the two versions of the DSME. The model migration executable, Δ_2, is an artifact of the evolution specification; that is, Δ_1 *describes* Δ_2. Once generated, Δ_2 will migrate the domain models as a standalone executable, or perhaps as a configuration file to a separate executable. It is important to note that Δ_1 is *not* used to evolve the DSME, but rather is a description of how the DSME *has evolved*. In other words, it is created with *a priori* knowledge of the DSME and DSME'.

The benefits of using a description of the DSME evolution as the basis for the model migration are plentiful. Some example benefits are,

- ability to use domain-specific concepts during specification,
- comparison between the two domain-specific modeling language definitions, and
- integration of domain-specific modeling language definitions with a common meta-metamodel.

Figure 19 also shows the conundrum of whether or not the executable system of the translated domain model should be equivalent to that of the original executable system. The executable system, being the incarnation of the programmer's intent, should certainly be consistent with the previous executable system, but the definition of consistency varies between evolutions of different DSMEs. Generally the *type* of

changes to the paradigm dictates the manner in which the domain models should be migrated.

Recall the two driving forces behind domain evolution: changes to the paradigm, and changes to the semantic domain. If changes are made to the semantic domain *only*, and the paradigm is preserved then the intent of the modeler should be preserved (i.e., the domain models will have the same meaning in the semantic domain, but after modifications to account for changes in the semantic domain). An example of this is the evolution of a model database that uses English measurement units to one that uses SI measurement units. Although no portion of the paradigm changes, the values of all unit-based attributes will not be correct in the semantic domain any longer, and should be modified.

If the changes are made to the paradigm *only*, and the semantic domain is preserved, then the intent of the models should be preserved (i.e., the domain models will have a different meaning in the semantic domain after modifications to account for changes to the CBS domain). An example of this is where the types of objects of a paradigm change, and they now have a different meaning in the semantic domain.

If changes are made to the paradigm *and* to the semantic domain, then some combination of these changes will be used. At this point, the modeler creating the domain evolution specification is actually assuming the role of domain modeler and revising the intent both of the modeler, and of the models, to reflect the new CBS.

The rest of this chapter focuses on the language framework used to specify domain evolution specifications, and does not account for the reasons why domain

evolution would take place. However, these concepts of the driving forces behind domain evolution will be used during the case study (Chapter V).

Justification for a Domain Evolution Framework

A meta-metamodel can be used to create a metamodel that describes any modeling language. That being said, not all meta-metamodels provide a convenient representation of the same abstractions, and some meta-metamodels, while very convenient for describing some abstractions, are tedious or counter-intuitive for others. This requires the concession (or, perhaps conjecture) that there is not now, nor will there ever be, one ubiquitous meta-metamodel. The consequence of this conjecture is that a *domain evolution tool* is required for each meta-metamodel. Although the complete set of domain evolution tools (one for each meta-metamodel) does not contain any identical tools, many of the tools will contain similar properties.

A *domain evolution framework* is a generic representation of the fundamental properties and algorithms that are contained in each and every domain evolution tool. As a generic representation it provides some parameters for instantiation, and those parameters are the meta-metamodel and transformation engine (e.g., graph-rewriting language) used to instantiate a particular domain evolution tool. This tool then serves as a domain-specific modeling environment that can be used for the evolution of any metamodels that are created using that meta-metamodel. When that domain evolution environment is used to evolve domain models from one metamodel to another, then a *domain evolution specification* has been created. The domain evolution specification must be specified in at least two dimensions: namely, the patterns in domain model *D* that are to be recognized and how those patterns should appear in the transformed model, *D'*.

The domain evolution specification generates a *model migration specification* that transforms the input model database into a migrated or output model database. The capability of graph-rewriting tools to perform this transformation lends them to usage as the model migration specification. However, given that individual graph-rewriting tools have different strengths and weaknesses when compared against one another, there may be different "best" implementations of a graph transformation for different styles of graphs. For example, a paradigm with significant levels of hierarchical containment would benefit from a graph transformation engine with efficient depth-first search algorithms. However, a paradigm with no containment would have no use for such a graph transformation engine, and would instead prefer an efficient breadth-first search algorithm. The problem is how to determine a methodology or tool that is customizable for different translation tools, yet allowing for the configuration of the translation in one central location.

The decoupling of the model migration specification from the graph-rewriting engine that implements that specification would release the specification from language-specific concepts, and more easily allow the translation of the model migration specification into more than one graph-rewriting language. Further, if the model migration specification were to be created with a language in the domain of model migration, then the creation of the model migration specification could be composed of concepts central to model migration, and the low-level details of the graph-rewriting engine could be delayed until the semantic translation phase. Just like any modeling domain, model migration has its own semantics and rules, and the ability to interpret a model migration language into a choice of semantic domains enables the modeler to pick

and choose what implementation of graph transformation to use, while at the same time keeping the transformation rules in model form.

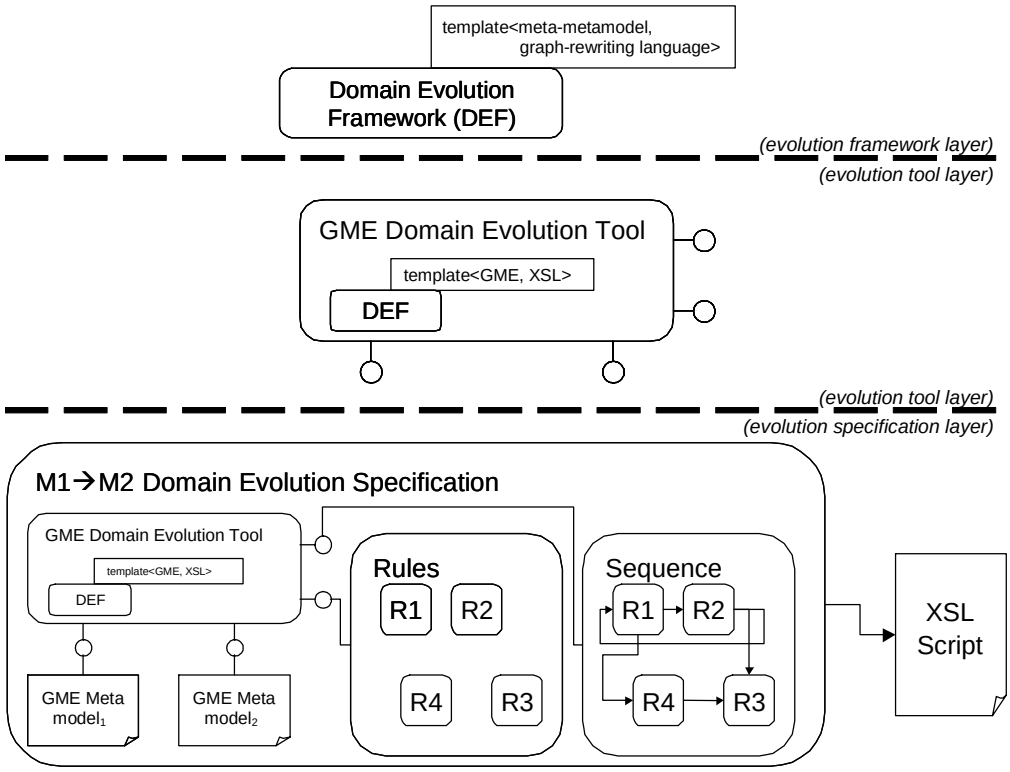


Figure 20. Layers of the domain evolution tool for MetaGME and XSL. Note that the final domain evolution specification is one particular evolution of domain models from GME Metamodel 1 (M1) to GME Metamodel 2 (M2)

Figure 20 is an illustration of these three layers for a particular instantiation of a domain evolution tool for the GME meta-metamodel and XSL graph transformations. The layers range from most generic at the top (the framework) to least generic (the specification that generates an XSL script). The *evolution framework layer*, at the top, contains the common information found in all domain evolution tools, and provides an interface to be customized by meta-metamodel and graph-rewriting language. In this sense, the evolution framework layer is quite similar to a template class definition in

C++. The *evolution tool layer*, in the center, has tools that are specific instances of the DEF. Shown in the figure is an example tool that parameterizes the DEF using the GME meta-metamodel and the XSL graph-rewriting language (of course, other combinations are possible). The domain evolution tool is an environment usable by a modeler who is performing model migration, and is roughly equivalent to an instantiated template class in C++.

The *evolution specification layer*, at the bottom, is used to create domain evolution specifications (Δ_1 in Figure 19) that generate a model migration executable (Δ_2 in Figure 19). Shown in Figure 20 is an abstract representation of what a domain evolution specification would look like using the domain evolution tool in the above layer. Evolution specifications are always specific to the particular DSME evolution, in this case, $M_1 \rightarrow M_2$. The GME Metamodel_1 and GME Metamodel_2 represent the current and evolved DSMEs, and the rule definitions (specification of the difference between the DSMEs), along with their sequencing, complete the domain evolution specification. This is roughly equivalent to the usage of an instantiated templatized class in C++, as the class exposes member functions through an interface (the two meta-metamodels, rules, and sequencing in Figure 20). These four parts of the interface are explained in detail throughout the remainder of this chapter.

This design for a framework has several appealing features. First, the look and feel of the user interface for defining rules and sequences will always be the same, regardless of the meta-metamodel used, as these interfaces are common across all meta-metamodels. Second, the rules used to migrate the domain models are created using a well-defined model of computation capable of being translated into the appropriate graph

transformation language. Finally, the common properties of the domain framework can be compiled into a library of existing code and operations that can be leveraged during the development of instantiated versions of the framework (i.e., tools). The remainder of this chapter is devoted to the definition and semantics of domain evolution framework concepts.

Overview of Components

The domain evolution framework is a collection of the common architecture components, user interfaces, programming interfaces of any tool that could be used to perform domain evolution. Any specific instance of the framework describes has four distinct parts: (1)(2) the two domain-specific modeling language definitions (as defined by a common meta-metamodel), (3) transform rules and their sequencing, and (4) the item types that can be matched along with the types of mappings that can be performed between matched items. Figure 23 shows an overview of this framework, with the four components identified.

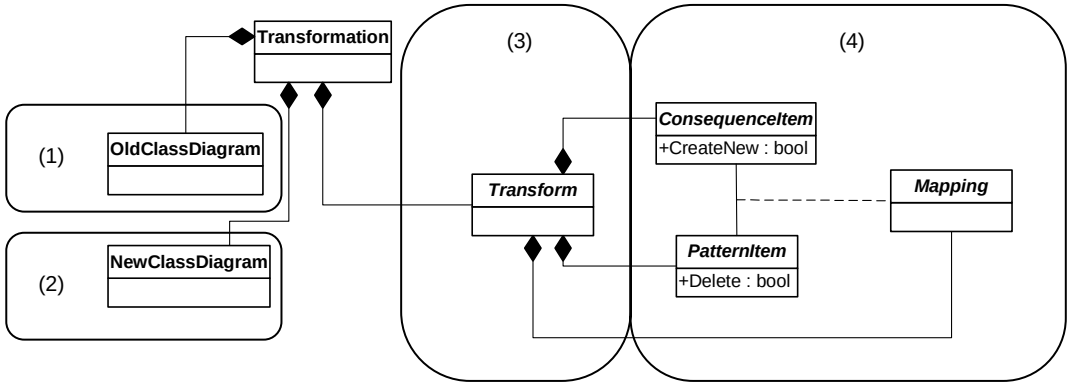


Figure 21. Simplified overview of the domain evolution framework. Comparison of this figure with Figure 20 shows the interfaces required for a domain evolution specification to be created with the framework

Justification of each of these components, as well as their definition *inside the framework* is provided in the remainder of this chapter. Note that the definitions given in this chapter are for the framework, and that supplemental definition is required to actually instantiate a particular tool.

Domain-Specific Modeling Language Definitions

Without a formal definition of the domain-specific modeling language (before, and after, its evolution), it is not possible to provide a formal description of how to evolve the models to conform to the evolved language. Moreover, the transformation may need to understand which language is the original, and which is the evolved.

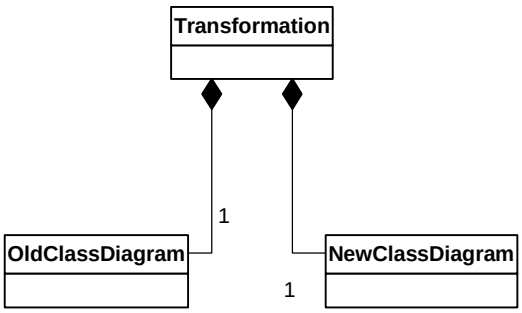


Figure 22. The transformation requires the formally defined "old" and "new" metamodels

Figure 22 shows how the transformation depends on the original and evolved domain-specific modeling languages. In this diagram, the original class diagram is referred to as the "old" language, and the evolution as the "new" language. Both language metamodels are required to have the same metamodel (i.e., all domain models must have the same meta-metamodel) in order to guarantee that there is a common storage format. In the framework, the formal description of the domain-specific

modeling language abstract syntax (and static semantics) is assumed to be defined in a form similar to that of a UML [5] class diagram (hence, the class names in Figure 22).

Of course, it is not required that the formal language definitions be in graphical form, or even in the form of UML. However, for the purposes of this research, a UML class diagram depiction is assumed. This is a viable assumption because even if a domain-specific language is described in some other notation, a UML class diagram could always be created for the purpose of evolution. UML class diagrams are used to describe the types of data, and a diagram could be created that described the data of the domain-specific language – if the model database(s) were of a sufficient size to justify the investment of time and effort to create such a diagram. As it will be discussed in a future section, the actual implementation of a model migration tool is dependent on the type of language used to describe the domain-specific modeling language – the meta-metamodel.

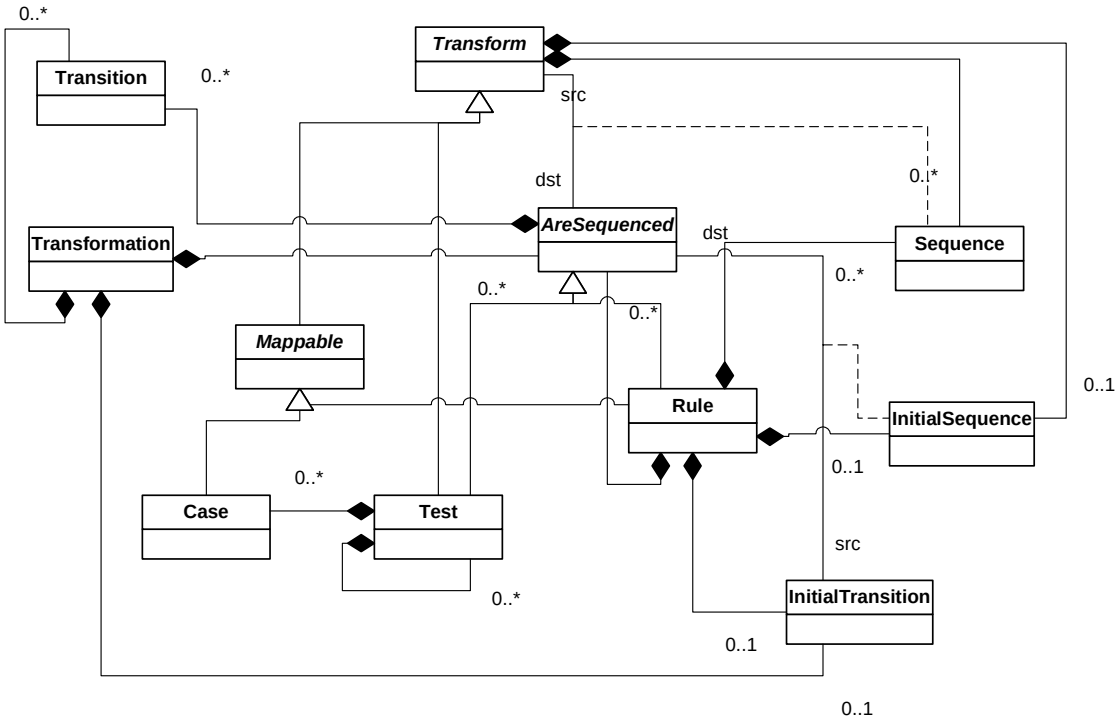


Figure 23. The class diagram of important elements of the transformation layout

Transformation Layout

The second portion of the framework definition revolves around the definition of the transform rules, and how they are sequenced (motivation for the necessity of sequenced rules is provided in a later section). The ordering of `Transforms` is imperative to provide the guarantee of determinism of the `Transformation`. The model of computation should allow for determinism not necessarily because determinism is required when migrating models, but rather because if determinism can be guaranteed then algorithms to migrate the models can be drastically simplified, and libraries of procedural algorithms can be referred to when designing a model migration solution.

The domain evolution framework prescribes the method of creating a set of transforms that perform modifications to portions of the input graph, and sequencing

these rules to achieve the actual migration of the domain models. The layout of these transforms, as well as their associations, is defined in the *transformation layout*.

The UML class diagram in Figure 23 represents an architecture that allows for deterministic transformations through the sequencing of transforms. The modeling concepts of inheritance and containment play significant roles in the definition of this portion of the domain evolution framework. This section gives detailed information of the inheritance hierarchy of the most used object types.

Transform Types

The two most important portions of this diagram are the `Transformation` class and the abstract `Transform` class (appearing in the upper-left area of Figure 23). In general the `Transformation` is made up of sequenced `Transforms`. `Transforms` are used to describe the specific differences between metamodels, but not all `Transforms` carry the same semantics. Fundamentally, not all `Transforms` modify the patterns they discover. Sometimes, the discovery of a pattern requires a side effect *somewhere else* in the `Transformation`. This gives rise to two mutually exclusive types of `Transforms`: those that perform mappings (i.e. have side effects) and those that do not.

However, there is another set of criteria for the existence of a solution: sequencing. In addition to their ability to describe the differences between metamodels, `Transforms` should also be capable of specifying where they should execute in control flow – their sequence. `Transforms` are not necessarily required to be sequenced, which leads to a second set of mutually exclusive types of objects: those that are sequenced, and those that are not.

The expansion of the two sets of types (side effects/not, and sequenced/not) do not specify a full Cartesian product of the two sets, because one combination does not make sense. This instance is the case of a not sequenced, side effect free type. It does not make sense for an object that is not sequenced to also not be able to map anything – this object would be excluded from sequenced consideration (i.e., it could not take part in the decision of control flow) yet it would not have a default behavior for the transform, as it could not map from one domain model to another. Therefore this type is left out, leaving three possible types which are explained below.

- `Rule` – has side effects, and is sequenced. It may also contain a sequenced object (either another `Rule` or a `Test`) in order to allow hierarchically structured `Transforms`.
- `Test` – is sequenced, but does not have side effects. It is used to make decisions in the sequence of `Transforms` by executing the `Cases` and other `Tests` it contains. Although it does not have side effects, the `Cases` it contains may.
- `Case` – is not sequenced, but has side effects. It is a special kind of `Rule` that behaves the same, but without the ability to be sequenced or contain any `Transforms`. Its purpose is to perform its mapping (if any) and return a boolean result if its pattern is matched. The value of the result is used for decision making in the execution of the model migration algorithm.

Legal Items

The replacement of syntax patterns from the old domain with patterns in the new domain is the basis of domain evolution. The types of objects that can be used to create these patterns are termed *legal items* in this paper, and depend on the language used to create that domain-specific modeling environment – the meta-metamodel. This portion of the evolution framework deals with those types of items allowed as patterns in a framework.

Consider, for example, a metamodeling language such as UML. Language ontologies defined using UML are defined in terms of classes and stereotypes, associations and composition, attributes and methods. When the language is changed, the new language is still defined using these same types. Regardless of the metamodeling language, this will always be the case: that the evolution specification will be defined using types that are directly dependent on the domain-specific modeling language's meta-metamodel.

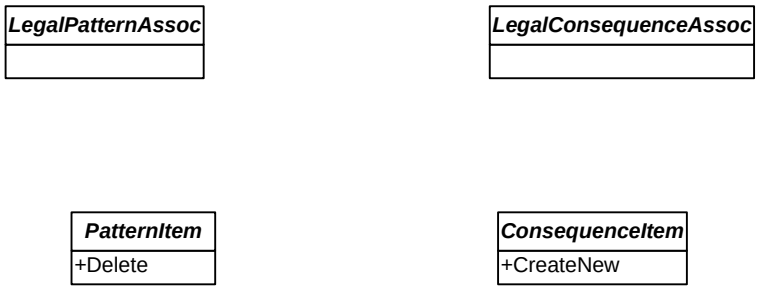


Figure 24. Legal items (without meta-metamodel definitions defined)

Figure 24 introduces four abstract types that, when specialized provide the archetypes for use in the patterns that are to be matched in the `Transforms`. All four of

these types are used in the next section dealing with the behavior of the individual transforms. The purpose of these types is to serve as an interface to the transformation specifications themselves.

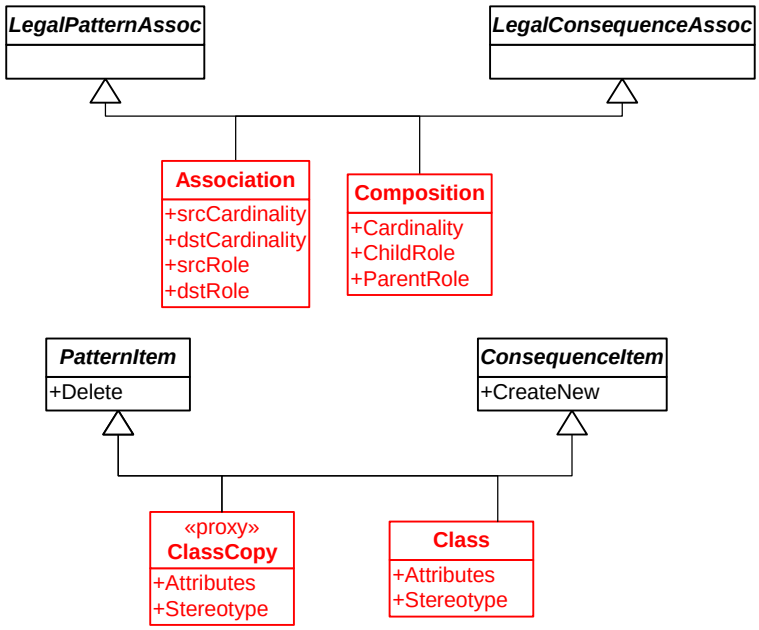


Figure 25. Legal items specification for the UML class diagram metamodeling language

Figure 25 is an example of the legal items specification for a simplified UML metamodeling language. The key elements are the `Class` and `ClassProxy` (a reference to an existing class, used as a shorthand across class diagrams). The key associations between these elements are `Associations` and `Composition`. The concepts of a proxy and a class may seem like "implementation details" of the metamodeling language. That is exactly what they are, and without capturing these implementation details it would be impossible to migrate models that were *defined* with those implementation details as part of the nomenclature of the metamodeling environment. The final result is that languages

defined using these UML concepts will be able to be evolved using the same concepts. The PatternItems all receive an attribute that prescribes whether or not to delete the item when a match is made. The ConsequenceItems have a similar attribute, but for creation rather than deletion.

Transformation Specification

Transforms may be sequenced and contained: containment allows Rule hierarchies. However, each type of Transform performs a different role upon execution – that behavior is the purpose of the transformation specification portion of the framework. The Test Transform performs only Parameter passing and flow control. The other two Transform types (Rule and Case) perform matching, based on their mixin type of Mappable.

The Mappable class is the center of the TransformationSpecification diagram, shown in Figure 26. Although six classes are shown in this diagram, the number of types of actual entities that can be used in a transformation specification is somewhat misleading. First of all, PatternItems and ConsequenceItems are abstract classes dependent on the type of meta-metamodel in use (recall that the "old" and "new" DSMLs must be expressed using in the same meta-metamodel). Secondly, PatternItems and ConsequenceItems are items of the same type, they merely play a different role in the Mappable object when instantiated.

Recall that the LegalPatternAssocs and LegalConsequenceAssocs are the abstract base types for the associations in which PatternItems and ConsequenceItems (respectively) may participate with each other. The transformation specification is a

collection of `PatternItems`, associated with each other, as they would normally be in a class diagram, which represents a particular syntactical pattern. The graph-rewriting engine searches the original domain models for this syntax pattern, and using the `Mapping` types of associations transforms `PatternItems` into `ConsequenceItems`.

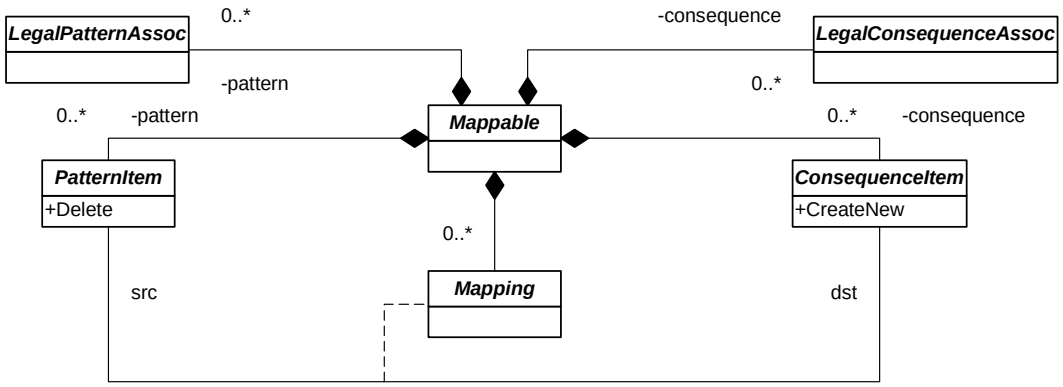


Figure 26. The Transformation Specification layout

The types for mapping associations form a fundamental set of operations, in similar fashion to the fundamental set of string replacement operations (i.e., insert, concatenate, delete, replace). The determination of the fundamental set of replacement operators for models is a significant problem in its own right. Textual replacement has only one "association" – position – while object replacement in a graph can have as many associations as are permitted for that type of object by the metamodel. This dissertation limits that set of operators to `CreateNew`, `CreateWithin`, `Becomes`, and `Delete` (analogous to insert, concatenate, replace and delete for textual manipulation).

- `CreateNew` – an attribute of a `ConsequenceItem`. If the LHS is matched then any object with this attribute set to true will be created. There must be a

CreateWithin association in which this ConsequenceItem is either the source or destination.

- CreateWithin – a binary association between a parent LegalItem (destination) and child LegalItem (source). The parent and child may be either a pattern or consequence, but at least one of them must be a consequence object. An object taking part in this association may not take part in a Becomes association, and may not have its Delete attribute set to true if it is a PatternItem.
- Becomes – a binary association between a source PatternItem and a destination ConsequenceItem. The source may not have its Delete attribute set to true, and the destination object may not have its CreateNew attribute set to true.
- Delete – an enumerated attribute of a PatternItem. If this attribute is set to ObjectOnly, and the PatternItem is part of a match then that item will be deleted (i.e., will not be copied directly into the output model) and any children of the object will be preserved by default and placed in the parent of the deleted object. If this attribute is set to ObjectAndChildren then all children will be recursively deleted.

The assumption that allows for this limited set of mappings relies on the definition of additional associations in a metamodel. Since these associations exist because of the metamodel, then those associations may be treated as objects – and created, replaced or deleted in their own right. While it may be possible to define more high-level mappings that ease the burden of the transformation, at this time no further

mappings are defined. It is important to note that the `Create` and `Delete` mapping associations are attributes of the objects that are to be created and deleted, rather than binary mapping associations. This is because during creation and deletion the source and destination endpoints (respectively) of a binary association would not exist.

Model of Computation for the Framework

The framework is designed to use any full-featured graph-rewriting engine as an execution tool. Since rewriting engines have different models of computation, it is important to define the strict execution semantics of the `Transform` classes, their sequencing and the passing of parameters so that the `Transformations` can be transformed into the artifact required by the graph-rewriting language. An independent model of computation removes the framework from association with only one type of graph transformation language, and allows for a mapping of its model of computation onto that of the preferred semantic domain.

Order of Execution

Generally, the order of execution is specified by linking two `AreSequenced` type objects with an ordered binary association. The handling of execution order is somewhat similar to that of StateCharts [91][92], where the type State is used instead of type `AreSequenced`. The specific rules and syntax for the sequencing are given in the following paragraphs.

A Transformation consists of one or more Transform type objects. When more than one object exists, then the first to execute must be specified. This is done by associating the `InitialTransition` object (which has a semantics similar to the initial

transition in StateCharts) with the Transform that should execute first (using an InitialSequence type association). There must not be more than one InitialTransition, and if there are more than one Transform objects in the Transformation, then there must be exactly one InitialTransition, and it must be the source of exactly one InitialSequence association.

To order the execution of other Transforms the Sequence association is used. Sequence associations can exist only between objects of type AreSequenced (Tests and Rules). The Sequence association is a binary association with two endpoints: the source, and the destination. In this directed association control flows from the source to the destination when the Sequence association is taken. If there is more than one AreSequenced object in the Transformation then each AreSequenced object must be either the source or the destination of a Sequence association. Thus, unlike the InitialSequence type association there can be more than one Sequence type association in a Transformation.

Control Flow

Control flow is managed using the Test type object. Since a Test functions similarly to the C 'switch' statement, it is possible to direct the flow of execution by taking an execution path only if a Case is true (recall that a Test contains 1..* Cases).

The difference between a Test and the 'switch' statement is the way in which the cases are specified. In the switch statement, a case is a value that must exactly equal the value that is found in the switch specification. In order for a case to be true (in a C-

style statement), the value must be matched *exactly* with that of the variable passed to the `switch`.

The `Test` type operates by comparing each value not as an exact match to a variable, but instead by the '`true`' or '`false`' return value of the `Cases` inside. The code found in Figure 27 is an abstraction of this behavior.

```
Case A { /* matching information */ }
Case B { /* matching information */ }
...
Case N { /* matching information */ }

Test {
  if A.match( )
    /* take specified control path */
  if B.match( )
    /* take specified control path */
  ...
  if N.match( )
    /* take specified control path */
}
```

Figure 27. Test statement as described in the style of the domain evolution framework (representative, but not actual syntax). Note that the Cases are defined to either match or not, so they are essentially a boolean result

Note that this example is not exclusive as to the control path that can be followed. It is possible to configure a `Test` to be *exclusive*, which means that in the abstract example above the '`if`' statements after `A.match( )` would be '`else if`' statements. A default execution path may also be specified (only applicable to exclusive `Tests`) which functions exactly as the default keyword in a C-style '`switch`' statement. This means that `Tests` not configured to be exclusive will be capable of following a control path for *each* `Case` that it contains.

The execution semantics gives no guarantees of determinism for picking a `Test` for evaluation. It is also up to the migration designer to create a mutually exclusive `Test`. This is because some evolutions may require multiple consequences for a particular pattern matched (for instance side effects may be required for all types, as well as different consequences for some specific types). The evolution framework must be capable of directing multiple paths to be followed from a particular portion of the input graph.

CHAPTER IV

DOMAIN EVOLUTION TOOL FOR METAGME POWERED BY XSLT

The specialization of the DEF is a specific domain evolution tool. This tool provides a domain-specific interface for creating model migration solutions. As previously discussed, the specialization occurs along two parameters – the meta-metamodel, and the graph-rewriting language. The case study examined in Chapter V is an example of model migration of GME metamodels, so the GME metamodeling language – MetaGME – is one axis of specialization presented in this chapter. The second is XSL – chosen because GME domain models are available in XML format. XSL is also well-defined for transformation across existing runtime engines, so the implementation of the XSL stylesheet can be performed by the preferred execution engine of the user and not be limited to one implementation of an XSL engine.

The domain evolution framework presented in the previous section is a useful framework for migrating domain models. However, to be useful as a domain-specific modeling language of its own it must be *specialized* to comprehend a particular meta-metamodel and to produce a particular graph-rewriting specification. The *specialized* domain evolution framework (now called the domain evolution tool) is then a domain-specific environment that may be used to migrate domain models. Once specialized, the domain evolution tool uses elements of two metamodels and associates them with one another using the domain model migration patterns and mappings found in the `Transformation` specification. The process to specialize the framework involves providing the metamodeling types that are to be used as `PatternItem` and

`ConsequenceItem` (and likewise for `Pattern` and `Consequence` associations). The details of these specializations are given throughout the rest of this chapter.[2]

The GME metamodeling environment

The Generic Modeling Environment (GME), developed at Vanderbilt University [100], is a meta-programmable visual modeling tool. As such it may be configured at design time to provide a domain-specific modeling environment through a metamodel [101]. The default metamodeling environment uses stereotypes and visualization standards that are representative of the fundamental design of core technology found in the MultiGraph Architecture (MGA) [3]. That is, they are the actual types that MGA is designed to use. As such, they are not necessarily the same types that another tool might use if it were designed separately from GME. Nevertheless, their existence was a design decision, and their usage in GME justifies their consideration when performing domain evolution, as there are a significant number of model databases in which these are the fundamental types.

The Container for Metamodel Definitions

The domain evolution tool requires knowledge of metamodeling types – the objects used to establish patterns – but this information alone is useless without knowledge of where those types are defined. Context of the definitions (which tells the domain evolution tool from which metamodel individual types are defined) can be used

[2] It should be noted that in the unspecialized evolution framework is implemented in class diagram form, and that many of the classes used in the definition are proxy (i.e., reference) objects that do not refer to any class. In order to specialize the framework, these classes are redirected to actual objects in the meta-metamodel. Throughout this chapter when we say a proxy is "pointed to" or "set to" or "will refer to" an object, then we are stating that the proxy object is redirected to refer to the object in the meta-metamodel.

by the domain evolution tool to guarantee that no types from the old metamodel exist in the migrated software, or that a Transformation does not use types from the new metamodel as a `Pattern` of a `Rule`.[3]

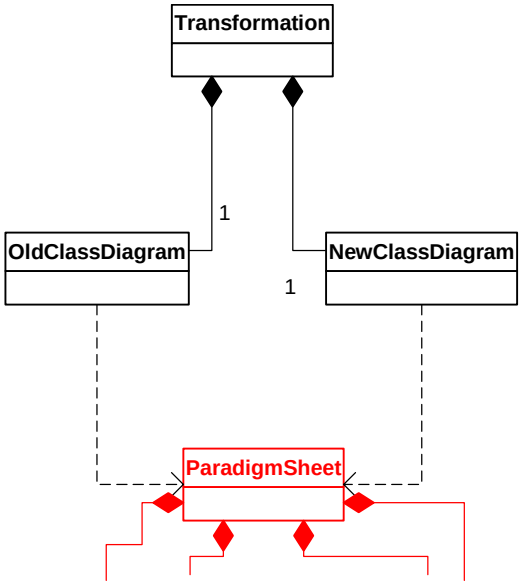


Figure 28. The `OldClassDiagram` and `NewClassDiagram` objects are pointed to the MetaGME `ParadigmSheet` object. This denotes that an object of type `ParadigmSheet` will be used to specify the `Transformation`. Once again, both metamodels are of the same meta-type

Figure 22 showed that a `Transformation` contains two collections – the old and new class diagrams. In order to specialize the `Transformation` to work with a particular meta-metamodel the container type for class definitions must be specified. This type is seldom the same for two different meta-metamodels (e.g., one meta-metamodel may refer to it as a `ClassDiagram` while another prefers `DiagramSheet`), so the generic `Transformation` definition cannot know in advance the specifics of the meta-metamodel

[3] As previously mentioned, a constraint that the `Patterns` be always composed of 'old' metamodel types or that the `Consequences` be always of 'new' metamodel types is not required. However, this is often the case, and justifies the ability of the domain evolution tool to provide the user with notification that this constraint is violated.

it will be using. The MetaGME paradigm uses a class called `ParadigmSheet`, so the `OldClassDiagram` and `NewClassDiagram` objects in will refer to the `ParadigmSheet` object in the GME meta-metamodel.

Patterns and Consequences

The `ParadigmSheet` object is defined in order for the `Transformation` to have knowledge about the context of the `Pattern` and `Consequence` objects – the core objects of the domain evolution specification. `Patterns` and `Consequences` (as well as any associations between them) are the same *types* of objects; they merely play a different role when used to specify an evolution. Recall that when domain evolution is performed each domain-specific modeling language must be defined using the same meta-metamodel – in this case, MetaGME. Thus, the `Patterns` and `Consequences` must be able to use the same types of objects when defining the specification.

Whereas the old and new class diagram objects were specialized using reference redirection (see the previous section) `Pattern` and `Consequence` specialization must be performed using the inheritance concept. This is because different meta-metamodels have a different number of fundamental types that can be used to describe metamodels. Any fundamental type that a meta-metamodel defines must be available during the domain evolution phase, so a more appropriate specialization is to use inheritance (allowing an arbitrary number of objects to be of the same type) rather than specify a quantity of types as proxies and then directing those proxies to the actual types.

An example of inheritance specialization for the domain evolution framework was provided earlier, but is fully defined for MetaGME in Figure 29. Once again, note that

the objects from the GME meta-metamodel (`ProxyBase`, `Constraint`, `FCO`, `Attribute`, `Src`, `Dst`, `Composition`, etc.) are fully defined in the meta-metamodel, not in this diagram. Also note that these objects actually utilize *multiple inheritance* to be both `Patterns` and `Consequences`. The astute reader will be curious as to how an object that is both a `Pattern` and a `Consequence` (though the IS-A inheritance relationship) could ever be distinguished as a `Pattern` or `Consequence` when it is contained in a `Transform`. However, this IS-A relationship does not exist when a type is used in a specification, because the user is asked to choose a role for the type – either `Pattern` or `Consequence`. Thus a `Pattern` will not use a `CreateNew` attribute, and a `Consequence` will not use a `Delete` attribute.

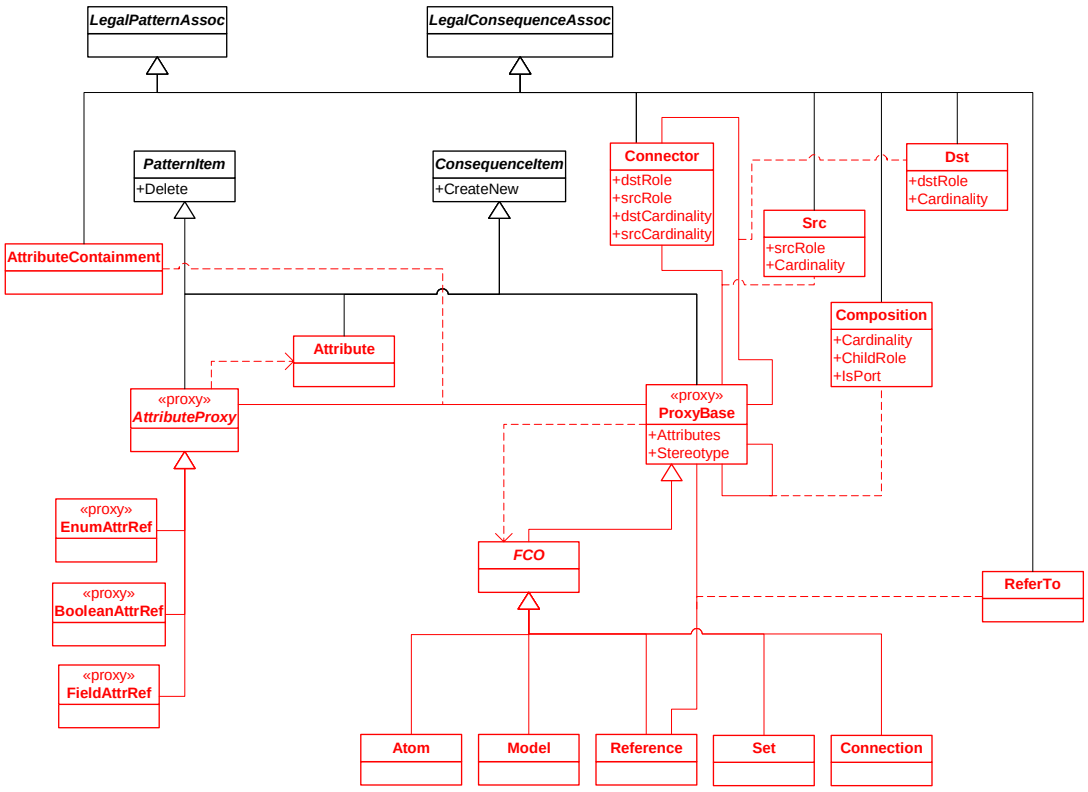


Figure 29. Specializing `Patterns` and `Consequences` for use with the UML paradigm

Mapping the DEF Model of Computation onto XSL

The previous chapter covered in detail the model of computation of the meta-metamodel independent DEF, and mentioned that its own model of computation could be mapped onto that of a full-fledged graph transformation language in order to implement the domain-specific semantics of model migration in that graph transformation language. This section will describe the mapping of the DEF onto XSL. First the XML storage format of GME is discussed, so as to determine the form of the XSL file, and then the model of computation mapping is provided.

Matching Patterns

The transforms created using a domain evolution tool must be translated into the semantic domain of a graph-rewriting engine before model migration can take place. The process of creating a semantic translator from the domain evolution tool to a particular graph-rewriting engine goes in two phases: first, the graph notation used to describe persistent models must be understood, and second the mapping from domain evolution pattern to graph-rewriting pattern must be formalized so that arbitrarily large patterns can be matched.

Therefore, in order to generate XSL that will traverse and modify models built according to a GME metamodel we must describe how data created using GME is stored. Refer to Figure 1 for the process of creating a GME metamodel and building data using that metamodel. The data is stored in accordance with the domain-specific XML schema (shown near the center of the figure). This domain-specific schema is generated from the specifications made in the GME metamodeling environment. Figure 30 shows an example metamodel, the schema it generates, and some example data conforming to that

schema. The colors and arrows show how the type of the object (and its attribute) are dispersed throughout the paradigm and data files.

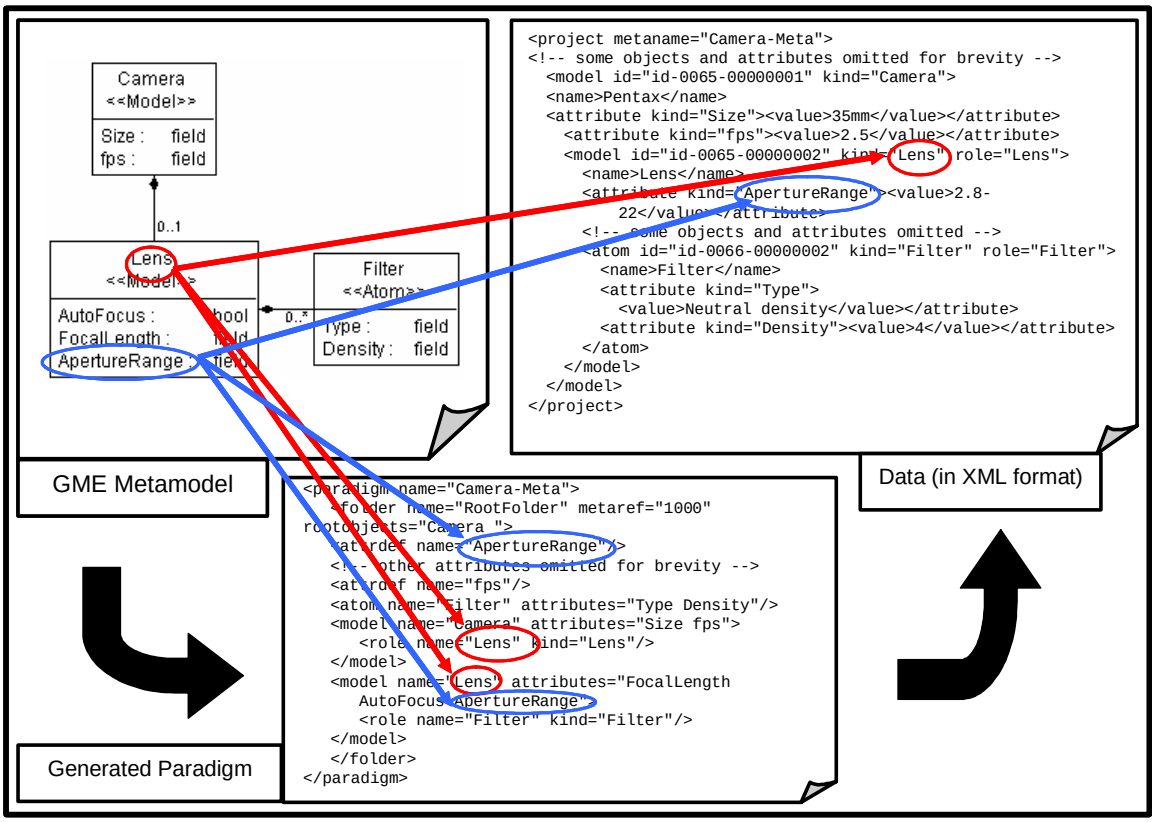


Figure 30. An example domain-specific modeling language created using the UML metamodeling environment. The red draws attention to object type and its dispersion throughout the paradigm and transform, while blue draws attention to the attribute type

Compare the structure of the previous figure with that of Figure 1 (p. 11). Note that the generated paradigm (implemented as an XML file that configures GME) uses domain concepts as elements and attribute names, and that those domain concepts are highly visible in any data created in the domain. The data artifact shows that the domain-specific concepts modeled in the GME metamodel are attached to instances in the data (these instances are the domain models). The method that GME uses to store its domain models should be explained further to avoid confusion.

The GME metamodel generates a paradigm which is an XML file (which will be referred to as the edf file) that conforms to a schema named `edf.dtd`. This edf file is used to configure GME at execution time, at which point GME becomes a domain-specific modeling environment where the domain is specified by the edf file. Domain models that are created using the DSME can be exported to an XML file (which will be referred to as the data file) that conforms to a schema named `mga.dtd`. The data file and the edf file have *no* relationship to each other in terms of well-formedness, but do share common types and attributes since they are both related to the same metamodel. This knowledge will be applied in future sections when the model of computation of the DEF is mapped onto XSL.

XML Representation in GME

GME is a graphical modeling environment that heavily relies on the concept of a multigraph to create models. GME uses XML format as one of its persistency models, which requires modification to the representation of some concepts in a multigraph architecture. The representation of a multigraph in XML structure (i.e., a tree) is possible only through the use of value-based associations that associate unique elements of the tree with each other across containment hierarchies. This is performed through the use of the ID and IDREF keywords in the XML schema that describes all GME model databases (see [100] for details).

The previous figure is an example of the formal mapping of the definitions in the visual language onto XML, using the knowledge of how a basic GME domain model is represented in XML format. The mapping for the representation of the domain models is shown below:

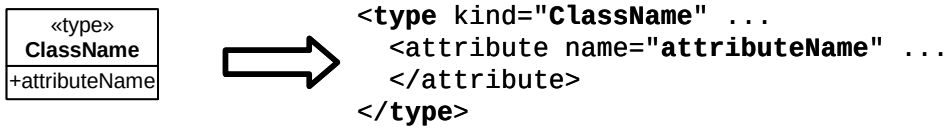


Figure 31. Formal mapping from a GME metamodel definition and the XML representation of an instance of that model created using GME.

Containment of objects is represented by actual containment in the XML document, as shown in Figure 32. Note that `FCO` would substitute for any first class object, but could never be instantiated, as it is abstract.

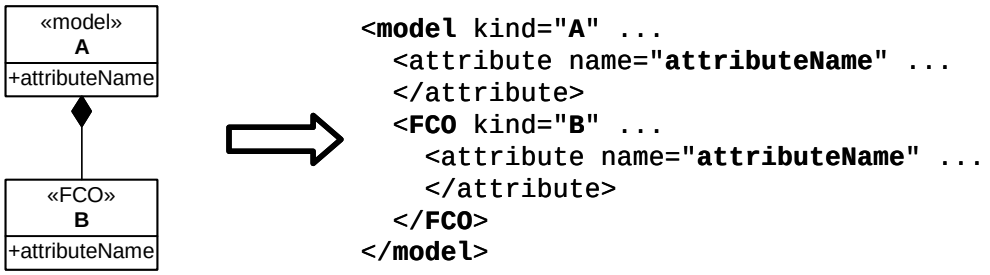


Figure 32. Formal mapping of hierarchy in a GME metamodel to an instance example of that hierarchy represented in XML format

Representation of associations (e.g., connections, sets, and references) is mapped out according to the example as given in Figure 33. Note that the `atom:D` does not contain any information that it is a member of the `connection:C` association, but that information is contained in the `connection` tag itself.

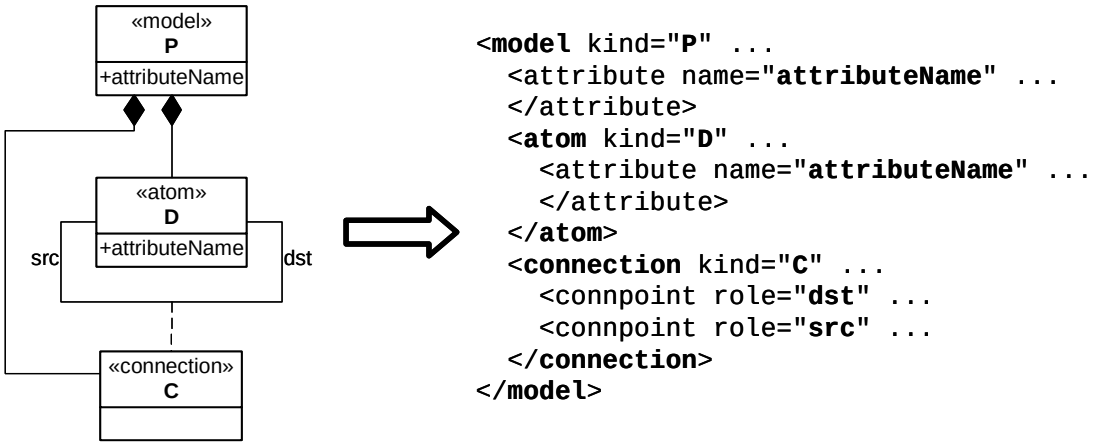


Figure 33. Formal mapping of association in a GME metamodel to an instance example in XML format

Other information is also encoded into the XML file, such as the paradigm name (and a global identifier to distinguish between different paradigm versions), position of the instantiated objects, icons that describe their appearance, and modifiers to specify other visual attributes such as the color. Some, but not all, of this additional information can be accessed for transformation using the language.

The Isomorphism

The XSL model of computation is conveniently defined from the perspective of model migration in that it is simple to create an identity transform, or *isomorphism*, for domain concepts that are not modified during domain evolution. The XSL model of computation (as described in Chapter II) prescribes that each node in the input XML document will be processed exactly one time, unless recursively called by a template. The document nodes are matched to the first template whose selection criteria they satisfy, and are passed to that template for processing.

XSL templates are sorted based on their level of complexity, where wildcard matches are considered the least complex out of all possible matches. Therefore, unless a particular object is specified to be changed, it will be processed by the isomorphic transform, shown in Figure 34. This template copies the current node to the output graph, and then attempts to match all children and attributes using the `apply-templates` command. This template is placed in every XSL stylesheet created by the GME model migrator tool. This presents a convenient method in which the isomorphism transform may be defined, because the isomorphism transform is the same for *all* nodes, due to use of the available wildcards in the XPath language (e.g., `@*`).

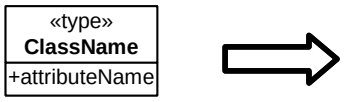
```
<xsl:template match="@* | node()">
 <xsl:copy>
  <xsl:apply-templates select="@* | node()"/>
 </xsl:copy>
</xsl:template>
```

Figure 34. The isomorphic transform used in XSL

Generated XSL

Given the formal definition of how instances of GME domain models are represented in XML format, it is possible to set forth a mapping from the patterns described using the model migration interface to patterns in XSL – or more accurately, in XPath. Each of the previous graphics in the examples that gave the storage layout of a particular domain model will be used to give the XPath expression that matches that particular instance pattern. In some cases, the XPath expression has been modified for readability.

For a generic object, the representation is as follows:



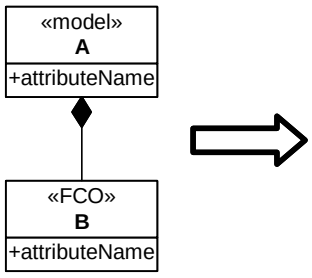

```
<xsl:template match="type[@kind='ClassName']">
  <!-- template body -->
</xsl:template>
```

Figure 35. Formal mapping from a graphical pattern in a transform and its XSL representation

Note that the `attributeName` attribute is not used in the XPath statement at all. This is because all ClassName objects will contain this attribute – otherwise, the objects would not be well-formed (a precondition of MM). Thus selection based on the containment of an attribute named `attributeName` is redundant. Because patterns are built from proxies to the actual objects, it is the name and type of the actual object that is used to create the templates, not that of the proxy object that appears in the diagram. The rest of the examples in this sub-section are all given with the actual objects as members of the pattern, rather than their proxies, to increase readability.

When containment is specified as part of the pattern the XSL selection process becomes complicated due to the fact that an exact match must be found, else the object should be copied. In the event of a complex containment match, it could happen that objects of the same type and in similar contexts might be treated differently depending on the circumstances (i.e., one might be directly copied, while the other is transformed). In the general case it is necessary to distinguish the *exact* circumstances of all elements relating to the current object being selected, called the *focus* in the transforms. As Figure 36 shows, the `FCO:B` is matched in all cases, but only transformed in the template if it has a parent of `model:A` type. If the parent does not exist (or in more complicated matches, if *any portion* of the pattern fails) then the object is copied to its output. Note that more than one `<xsl:when>` can be contained inside the `<xsl:choose>` tag, meaning that a

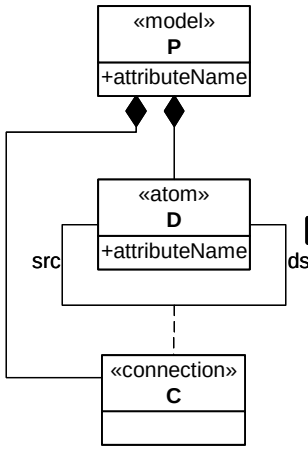
multiple number of possible tests involving FCO:Bs can be tried before the isomorphism is applied.




```
<xsl:template match="FCO[@kind='B']">
 <xsl:variable name="B4_A"
              select="parent::model[@kind='A']"/>
 <xsl:variable name="B4_focus_B"
              select="current()"/>
 <xsl:choose>
  <xsl:when test="current()[$B4_focus_B][$B4_A]">
   <!-- template body -->
  </xsl:when>
  <xsl:otherwise>
   <!-- perform isomorphic transform -->
  </xsl:otherwise>
 </xsl:choose>
</xsl:template>
```

Figure 36. Mapping a containment pattern into XSL

When associations are part of the pattern then the variables and tests similar to the case of containment is extended. The complicating factor here is that connections, references, and sets, can be contained *anywhere* in the hierarchy of the root object, so the entire document must be searched to find the associations of which an object is a participant.




```
<xsl:template match="atom[@kind='D']">
  <xsl:variable name="Becomes4_C"
     select="//connection[@kind='C']"/>
  <xsl:variable name="Becomes4_P"
     select="parent::model[@kind='P']"/>
  <xsl:variable name="Becomes4_focus_D"
     select="current()"/>
  <xsl:choose>
   <xsl:when test="current()
     [@id=//connection[connpoint[@role='src']
    /@target=connpoint[@role='dst']/@target]/
     connpoint[@role='src']/@target]
     [$Becomes4_C][$Becomes4_focus_D]
     [$Becomes4_P]">
    <!-- template body -->
   </xsl:when>
   <xsl:otherwise>
    <!-- perform isomorphic transform -->
   </xsl:otherwise>
  </xsl:choose>
 </xsl:template>
```

Figure 37. Mapping an association pattern (connection) into XSL

Figure 37 is an example pattern that uses an association as a portion of the context. Note that the unique identifiers are used to guarantee that the `dst` and `src` objects are in fact the same object, and that the connection is contained in the `model:P`. The same basic methods are used when a reference or set is used as part of the pattern rather than a connection, the only modifications being that the attribute names and hierarchy of the set and reference are different.

Mappings

The mapping of a matched object occurs in the body of the template. The default mapping, of course, is a direct copy into the output graph. When the template body is entered, it is passed a set of nodes that conform to the pattern of the template that that have satisfied all of the filters in any `<xsl:when>` tests. This set is then immediately

passed to a named template for further processing. Named templates are executed in an XSL on command, and do not have any matching statements.

For each mapping in the model migration specification a named template is generated in an XSL stylesheet. An example named template (and its corresponding diagram) of the `Becomes` mapping is given in Figure 38. Note that each named template is encoded with the change that it performs (in this case `B` becomes a `D`) and it is uniquely named to avoid conflicts with multiple changes of the same type. Containers and shading are added to the graphical specification and its generated XSL template to call attention to which portions of the XSL are generated from which areas of the graphical specification.

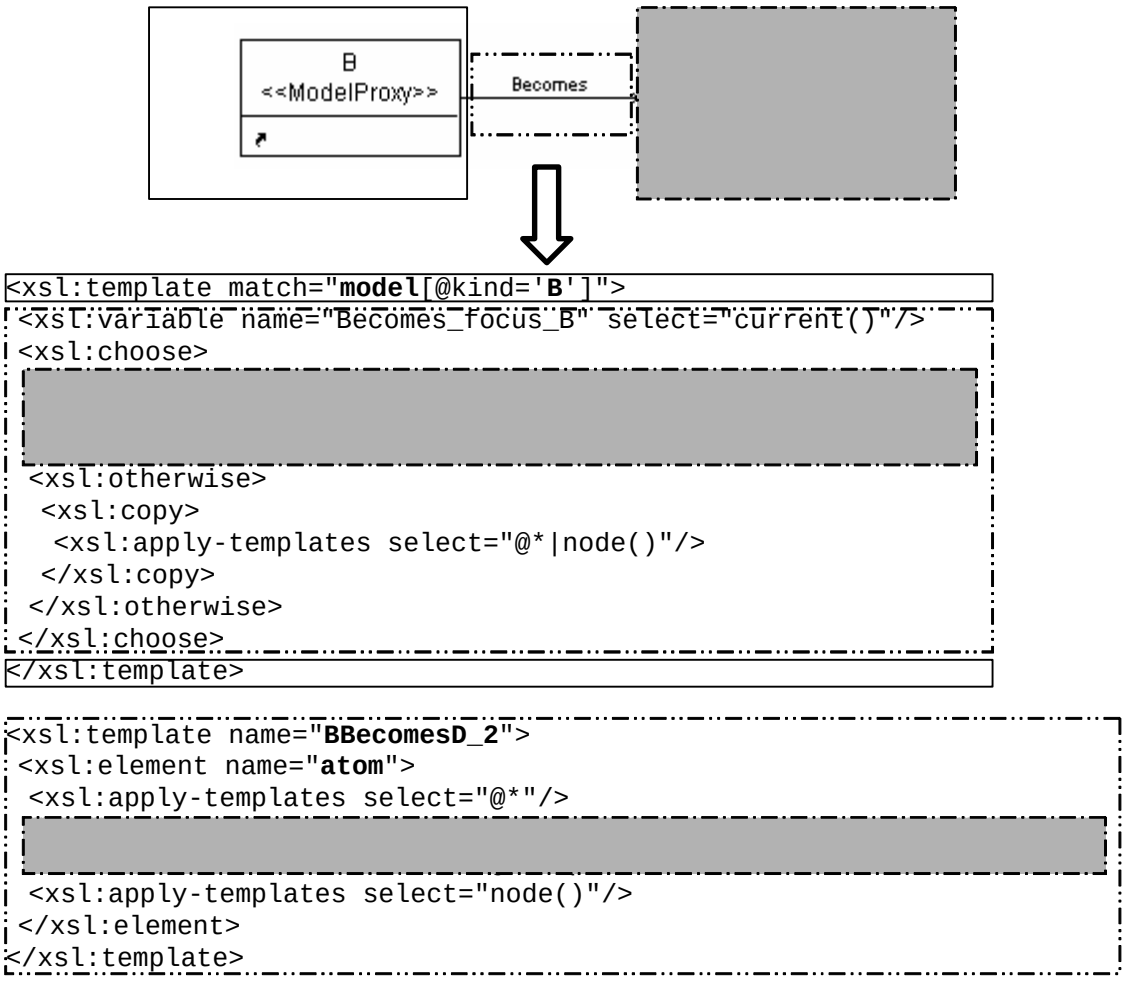


Figure 38. Mapping transformation specified as a named template

An analogous method is used when performing `CreateWithin` (and consequently `CreateNew`) except that the template would be called from within the template body of the object in which the consequence was to be created. When using the `Delete` mapping the template body is empty.

Sequencing

XSL stylesheets prescribe that nodes from the input document are processed in document order [33]. This provides a deterministic behavior between executions of the

same input, but not a predictable behavior given any arbitrary input. As such, it would be impractical to design a multi-pass algorithm in one stylesheet (that is, in one execution step) when a multi-pass approach could be used.

There are several ways in which this may be done. One method would be to use name-mangling to guarantee the uniqueness of mapping concepts and patterns (in the event that a pattern and mapping are used in different `Transforms`). However, this would still not work as identically matched patterns could not be distinguished by the XSL processor.

Another method is to take advantage of the *mode* parameter of a template, and have a different execution mode for each `Transform` in the overall transformation. However practical this may seem from an execution standpoint, it is decidedly impractical when considering debugging of algorithms, as a stylesheet of this nature behaves like a spaghetti code program, rendering it impractical to debugging.

A third solution is to create a different stylesheet for each `Transform` in the transformation, and pass an XML artifact to each stylesheet, keeping only the final XML artifact. This is the implemented solution, due to its guaranteed predictability of execution, and its easily composable nature. Also, by separating the execution into different XSL stylesheets, it becomes easier to debug certain portions of the algorithm by passing the XML document to that stylesheet only, and checking the output for errors. Figure 39 shows the sequencing execution model. This implementation is not optimized for performance at runtime, rather optimized for rapid generation of the XSL templates at interpretation time of the domain evolution specification.

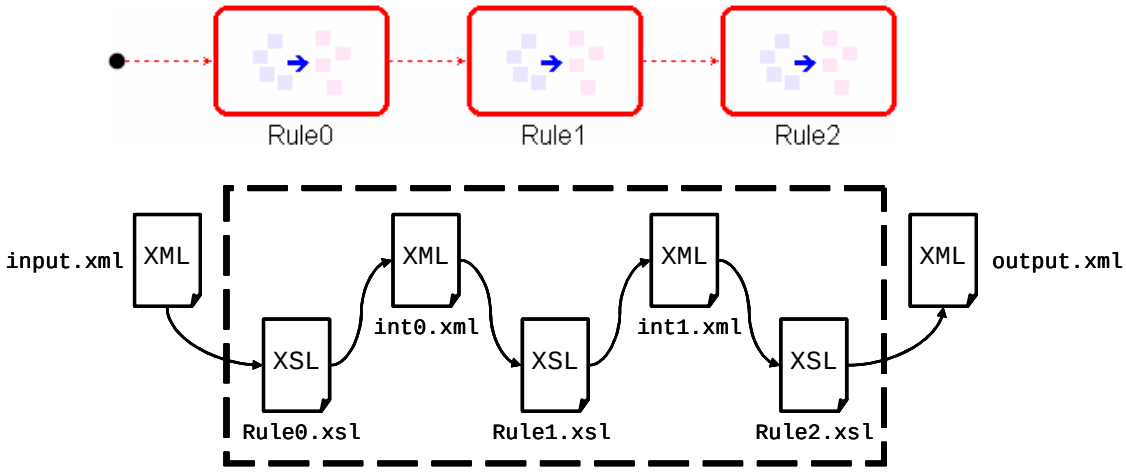


Figure 39. Example transformation exemplifying the translation into a sequenced execution

Tests and Cases

Implementation of the `Test` and `Case` types of `Transforms` requires extremely low-level direction of the flow of control using XSL file output. Since the execution of a Test requires an attempt to execute all contained Cases (but not in any explicit order), and since the control flow of those Cases is followed *only if the Case pattern is matched*, the XSL transformation engine tests the current input XML file for possible control flow continuation for the first `Case`. When that `Case` execution path is finished, the next `Case` is tested against the current input file, which is the *output* XML file of the previous `Case` execution path, rather than the input file used by that `Case`. An example diagram and control-flow output is shown in Figure 40.

As with the general execution implementation (shown in the previous section) this implementation method will have poor performance. Ideas for increasing the performance of this (and the previous) implementation methods are given in the final chapter.

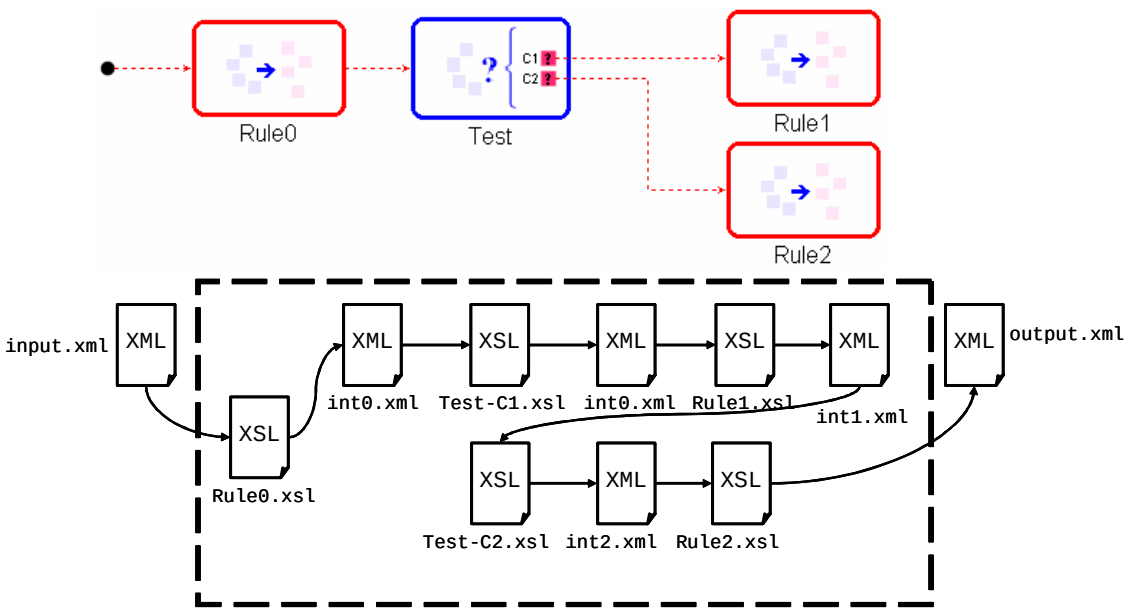


Figure 40. Execution example for Transformation with Test and Cases

Paradigm Name

The purpose of including the `ParadigmSheet` objects as part of the transformation process is for the migration of domain models from one paradigm to another. Frequently these paradigms have the same name, but on many occasions the names are different. As the XML file has the paradigm name encoded within it, this portion of the file must be changed in order to load the output XML file into the new paradigm. By adding references of the class diagrams to the `OldParadigm` and `NewParadigm` aspects of the `Transformation` model, the name of the `OldParadigm` reference is replaced by the name of the `NewParadigm` reference (note this is the name of the reference, not necessarily the name of the class diagram to which it refers, which may require user input after the creation of the reference). If the paradigm name does not change, then there is no need to include the references to the class diagrams.

Implementation Details

Much of the domain evolution framework's execution and model of computation is abstracted from the meta-metamodels and graph-rewriting implementation used, and is provided as a compiled library available to developers of domain evolution tools. This compiled library contains the execution of the interpreter, which performs the translation of the domain evolution specification by specifying visits to classes common to all tools (such as `Transforms` and `Ports`) as well as abstract classes (such as `PatternItems`). In order to take advantage of this abstract behavior, virtual methods and classes are used throughout the abstract implementation to allow for specialization of behavior and meta-metamodel definition independent of the algorithms for the model of computation of the domain evolution framework. The abstract framework utilizes the visitor design pattern [99] and – similar to the specialization of the language – is extendable through inheritance. In addition to the visitor traversal specification (i.e., the form), there also is the behavior of the visitor when it arrives at an appropriate node (i.e., the function). Form and function are explained and examined below, for the MetaGME and the XSL graph-rewriting engine.

Customized Node Classes – Form

The types of `Patterns`, `Consequences`, and the type of their container object (for MetaGME, a `ParadigmSheet`) determine the specialized form of the domain evolution tool. As previously mentioned, much of the semantics of the model of computation can be expressed in terms of classes that exist for all meta-metamodels – e.g., the `AreSequenced` and `Transformation` types. However each meta-metamodel will require its own specialized architecture to account for its specific `Pattern` and `Consequence`

classes. The specialization is accomplished by deriving meta-metamodel-specific classes from the abstract base class that defines the default behavior of a `Pattern` and `Consequence` – called a `LegalItem`.

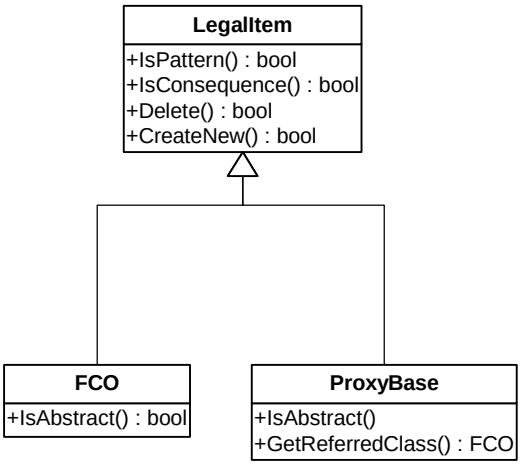


Figure 41. When building the class hierarchy for the interpreter, `FCO` and `ProxyBase` (types in the UML meta-metamodel) derive from abstract class `LegalItem`. This allows visitor classes to visit `LegalItem` nodes and perform transforms specific to this meta-metamodel through polymorphism in the `LegalItem` type

The `LegalItem` class is an abstraction for the use of the interpreter. It allows for inheritance of the meta-metamodel specific types from one class, and then provides for determination of whether objects of that type are `Patterns` or `Consequences` in the evolution specification. Figure 41 shows the GME meta-metamodel nodes and how they inherit from classes in the interpreter hierarchy.

Note that the `FCO` and `ProxyBase` types can provide their own instantiations of the methods of `LegalItem` as the methods are virtual, enabling polymorphic behavior. Also note that customized methods exist for the `FCO` and `ProxyBASE` types to determine whether or not they are abstract, as well as the `ProxyBase`'s ability to determine of which

`FCO` it is a reference. Omitted are the method names that exist for `FCO` and `ProxyBase` to gain access to the context of the object in its metamodel (e.g., containment associations of which it is the source or destination). The visitor classes that are specialized for a meta-metamodel often use these methods, as discussed in the next section.

Customized Visitor Classes – Function

Despite the specialization of the language for a particular meta-metamodel, the form of the domain evolution language is largely unchanged after specialization occurs. This allows for much of the algorithm for traversing domain evolution specifications to be encoded in a *paradigm independent* manner – that is, much of the traversal logic is independent of what types of `LegalItems` exist for a particular meta-metamodel. However, the behavior at traversed nodes of the graph is dependent on polymorphism to generate the correct graph-rewriting specification. Figure 42 shows a portion of the class hierarchy that exists for the GME and XSL specialization of the domain evolution framework, along with examples of other classes that could be created for use with other meta-metamodels and graph-rewriting engines.

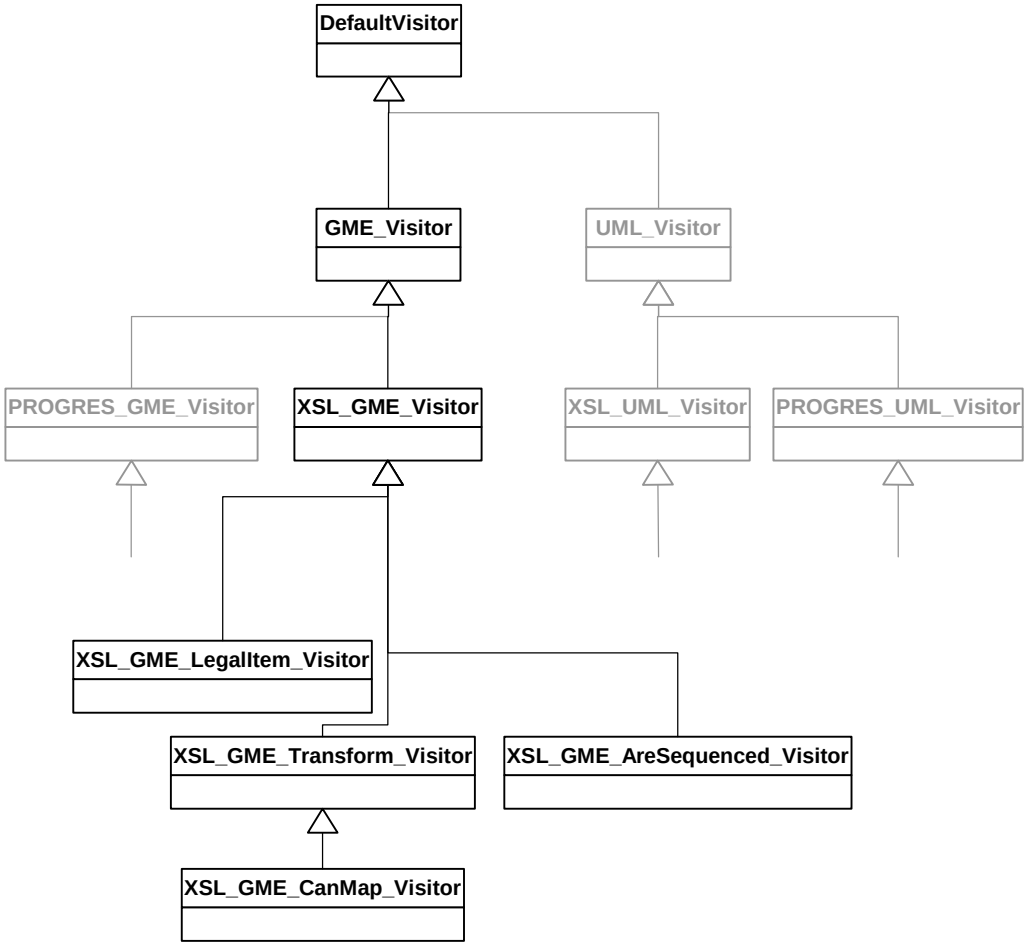


Figure 42. Class hierarchy for the traversal of nodes in the interpreter, utilizing the Visitor design pattern. Ghosted classes denote where other meta-metamodels may be incorporated in other designs, while the darkened classes show the specialization of the visitor – initially to the GME meta-metamodel, and then to the XSL graph-rewriting specification. Several classes are omitted for brevity

The domain evolution framework interpreter provides one abstract visitor class that is used in the visitor traversal algorithm. The `Transform` and `Transformation` classes contain lists of (or associations to) the meta-metamodel specific types used in the domain evolution specification. Use of inheritance of these specific types from the `Pattern` and `Consequence` types allows for the use of polymorphism among visitor classes to efficiently produce the semantic translation of the specification into the graph-

rewriting language. The `dynamic_cast` C++ keyword is used to eliminate problems with double dispatching.

In addition to creating the hierarchy of visitors, the implementation for individual `visit` methods must be performed. The rationale behind overloaded signatures for the `visit` methods is to provide a customized implementation for that signature type. The implementation of these methods will produce the XSL transformation – which requires analysis of the makeup of the data storage scheme described previously.

CHAPTER V

CASE STUDY

Two examples are presented in this chapter that will aid in the understanding of the language and in its ability to migrate domain models. The first of these is primarily presented as an academic exercise of an archetypal modeling evolution problem. The second is an actual usage of the model migration paradigm to migrate models used by an industry partner.

Evolution through Specialization of Domain Concept

A canonical example of the requirement for domain evolution is found in the specialization of a domain concept into two or more derived concepts. For example, consider the two metamodels shown in Figure 43. In the domain of signal processing, there are individual signal processors that may be joined together through their `Ports` via `Connection` associations. However, the domain language evolved by requiring that there be exactly two different types of `Port`, `Input` and `Output`, and furthermore, that `Input` and `Output` ports may be connected only in accordance to certain static semantics, namely the following.

- Any `Port` may not be the source of a connection where the destination of the connection is itself (previously existing constraint)
- An `Input` may not be the source of any connection in which the destination of that connection is an `Output`

- An `Input` may be the source of a connection in which the destination is an Input provided that the parent `ProcessSignal` of the destination is contained by the parent `ProcessSignal` of the source
- An `Output` may be the destination of a connection in which the source is an `Output` provided that the parent `ProcessSignal` of the source is contained by the parent `ProcessSignal` of the destination (i.e., the connection serves as a pass-through that crosses hierarchy boundaries)

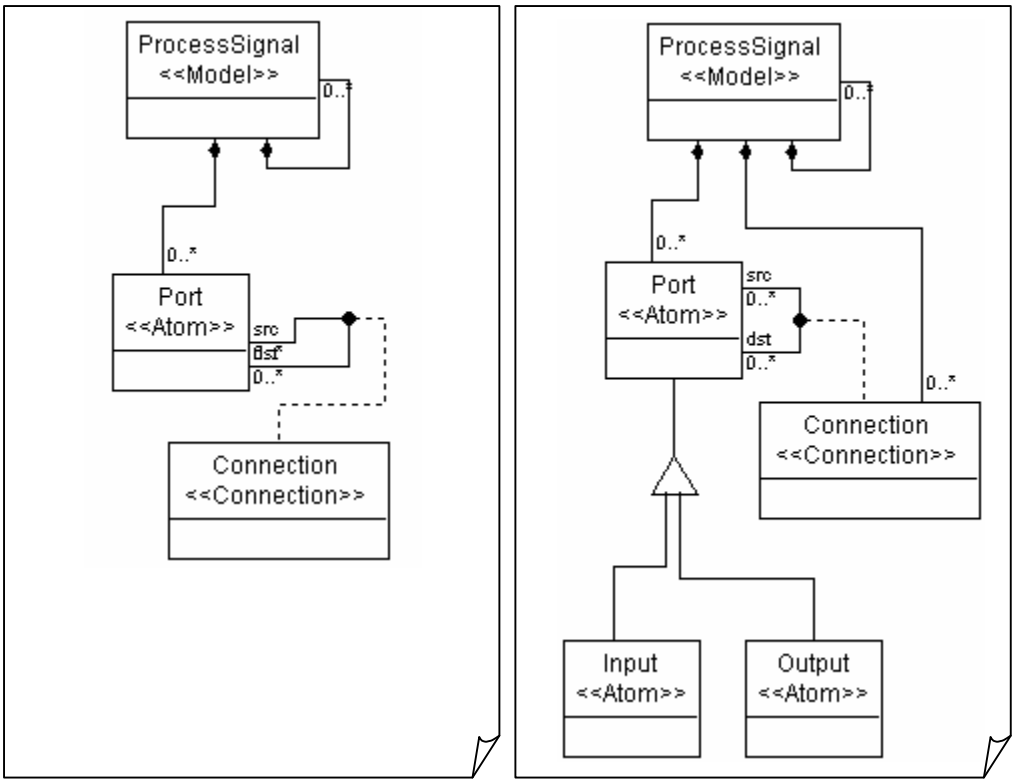


Figure 43. The original metamodel (left) and the evolved metamodel (right)

Simply said, `Input` and `Output` ports may be used to pass along the input or output signals of parent or child processes, but should never connect the same type if the `ProcessSignals` exist on the same level of hierarchy.

Algorithm

In this case, we wish to maintain the exact semantics of the domain models when they have been evolved, so we assume that the original modeler intended that the `Port` objects behave as if they were the `Input` or `Output` ports in the evolved paradigm. Given the existing constraints, and the evolved constraints, it is possible to write an algorithm to convert any input domain model into its equivalent evolved domain model, while maintaining the intention of the original modeler. The algorithm is as follows:

1. Transform all `Ports` that will become `Input` ports connected to other `Input` ports
2. Transform all `Ports` that will become `Output` ports connected to other `Output` ports
3. Transform all remaining `Ports` that will become `Input` ports
4. Transform all remaining `Ports` that will become `Output` ports

This algorithm is fairly simple, but must be divided into at least two portions, based on the fact that any port may be either an `Input` or an `Output` if it is the source of a connection, due to the ability of ports to pass along signals from `ProcessSignal` models on another level of hierarchy. Since this sort of control flow exists, the algorithm is divided into four sequenced `Transforms`, as shown in Figure 44.

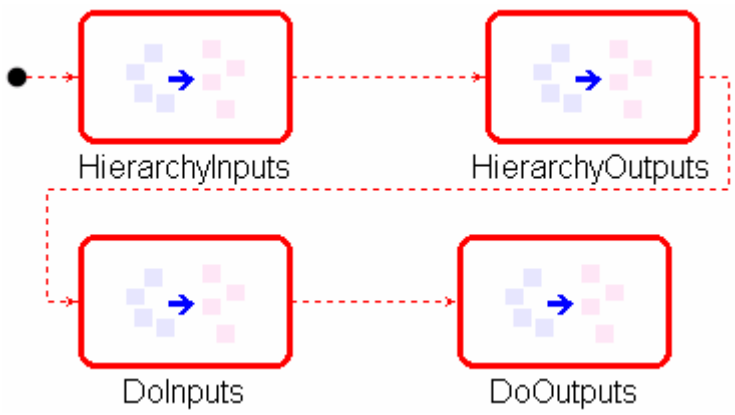


Figure 44. The sequence of `Transforms` to evolve the domain models

The domain evolution language provides interfaces to generate any type of graph-rewriting language that is convenient to the end user. In this case, the domain models are all stored in an XML format, which tailors nicely to the generation of XSL stylesheets to perform the evolution. Each `Transform` in the sequence above will generate an XSL document, which will be called in sequential order on the output artifact of the previous Transform output, until the final document – suitable for use in the new paradigm – is produced.

Transforms and Output XSL

Only two `Transform` contents need be shown, since the other two will be the dual (i.e., for `Outputs` instead of `Inputs`). The first of these is the contents of the `HierarchyInputs Transform`, which is shown in Figure 45. This represents the algorithm step number 1. Notice that the parent of the source `Port` is the grandparent of the destination `Port`. The output XSL is given in Figure 46 and Figure 47.

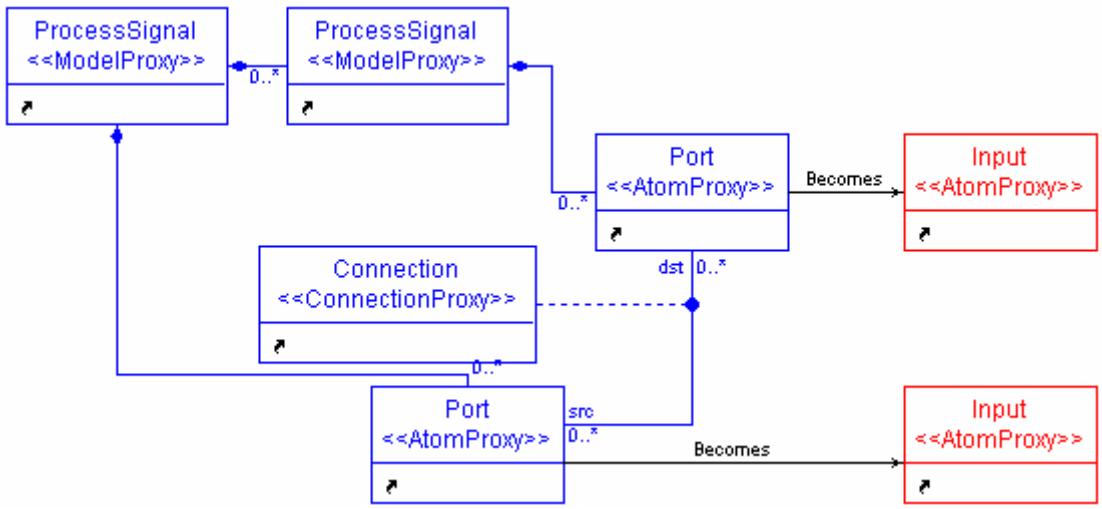


Figure 45. Contents of the HierarchyInputs Transform

```
<xsl:template name="PortBecomesInput_62081432">
  <xsl:element name="atom">
   <xsl:apply-templates select="@*"/>
   <xsl:attribute name="kind">Input</xsl:attribute>
   <xsl:attribute name="role">Input</xsl:attribute>
    <xsl:comment>
     transformed using template 'PortBecomesInput_62081432'
    </xsl:comment>
   <xsl:apply-templates select="node()"/>
 </xsl:element>
</xsl:template>
<xsl:template name="PortBecomesInput_62081992">
  <xsl:element name="atom">
   <xsl:apply-templates select="@*"/>
   <xsl:attribute name="kind">Input</xsl:attribute>
   <xsl:attribute name="role">Input</xsl:attribute>
   <xsl:comment>
    transformed using template 'PortBecomesInput_62081992'
   </xsl:comment>
   <xsl:apply-templates select="node()"/>
  </xsl:element>
</xsl:template>
```

Figure 46. XSL Output for the HierarchyInputs Transform (named templates)

Each pattern object is given its own context in relation to the objects being transformed through the `Becomes` association, and this context is shown in the `xsl:variable` that is created for each member of the pattern context. When each node is parsed in the input file, it is passed through the set of filters to determine whether it is

matched. If so, it is passed on to become an `Input`; if not, it is copied as-is. Calls are made to the named templates (seen in Figure 46) if a match is found (matching attempts shown in Figure 47).

```
<xsl:template match="atom[@kind='Port']">
 <xsl:variable name="Becomes6_focus_Port6"
    select="current()"/>
 <xsl:variable name="Becomes6_ProcessSignal6"
    select="current()/parent::model[@kind='ProcessSignal']"/>
 <xsl:variable name="Becomes6_Port6"
    select="current()/parent::model[@kind='ProcessSignal']
      /child::model[@kind='ProcessSignal']
      /child::atom[@kind='Port']"/>
 <xsl:variable name="Becomes6_Port62"
      select="current()/parent::model[@kind='ProcessSignal']
      /parent::model[@kind='ProcessSignal']
      /child::atom[@kind='Port']"/>
 <xsl:variable name="Becomes6_ProcessSignal62"
      select="current()/parent::model[@kind='ProcessSignal']
      /parent::model[@kind='ProcessSignal']"/>
 <xsl:variable name="Becomes62_ProcessSignal62"
      select="current()/parent::model[@kind='ProcessSignal']"/>
 <xsl:variable name="Becomes62_ProcessSignal620"
      select="current()/parent::model[@kind='ProcessSignal']
      /child::model[@kind='ProcessSignal']"/>
 <xsl:variable name="Becomes62_focus_Port62"
      select="current()"/>
  <xsl:choose>
   <xsl:when test="current()
     [@id=//connection[child::connpoint[@role='dst']
     /@target=$Becomes6_Port6/@id]
     /child::connpoint[@role='src']/@target]
     [$Becomes6_focus_Port6][$Becomes6_ProcessSignal6]
     [$Becomes62_ProcessSignal620][$Becomes6_Port6]">
    <xsl:call-template name="PortBecomesInput_62081992"/>
   </xsl:when>
  <xsl:when test="current()
     [@id=//connection[child::connpoint[@role='src']
     /@target=$Becomes6_Port62/@id]
     /child::connpoint[@role='dst']/@target]
     [$Becomes62_focus_Port62][$Becomes62_ProcessSignal62]
     [$Becomes6_ProcessSignal62][$Becomes6_Port62]">
   <xsl:call-template name="PortBecomesInput_62081432"/>
  </xsl:when>
  <xsl:otherwise>
   <xsl:copy>
    <xsl:apply-templates select="@*|node()"/>
   </xsl:copy>
  </xsl:otherwise>
 </xsl:choose>
</xsl:template>
```

Figure 47. XSL Output for the HierarchyInputs Transform (main stylesheet matched templates)

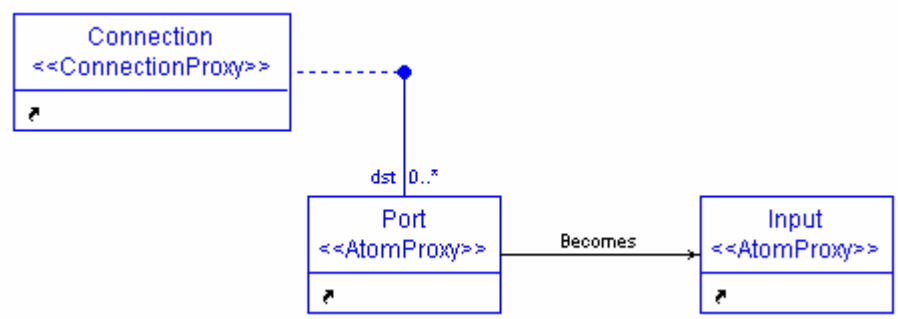


Figure 48. Contents of the DoInputs Transform

The other `Transform` contents that bear noting are those of the `DoInputs Transform`, shown in Figure 48. In this portion of the algorithm (step 3) the destination of the `Connection` and the containment context of the `Port` are irrelevant: only the type of the destination matters. Therefore, no further context is required for the `Transform` pattern, in essence, the connection type is a wildcard. The XSL output for the `DoInputs Transform` is much simpler, but omitted for brevity.

Domain Models – Before and After

Consider the set of models in Figure 49. The upper portion of the figure shows the high level of the models, and the lower portion is the contents of `ProcessSignal1`, which contains another level of `ProcessSignal`. Note that the contents of `ProcessSignal1` satisfy the hierarchy portion of the algorithm, where `Input` and `Output` ports may be connected to others of the same type. Also, note that the name "Port" on the objects is an instance name, not a type name (type in this domain is denoted by icon).

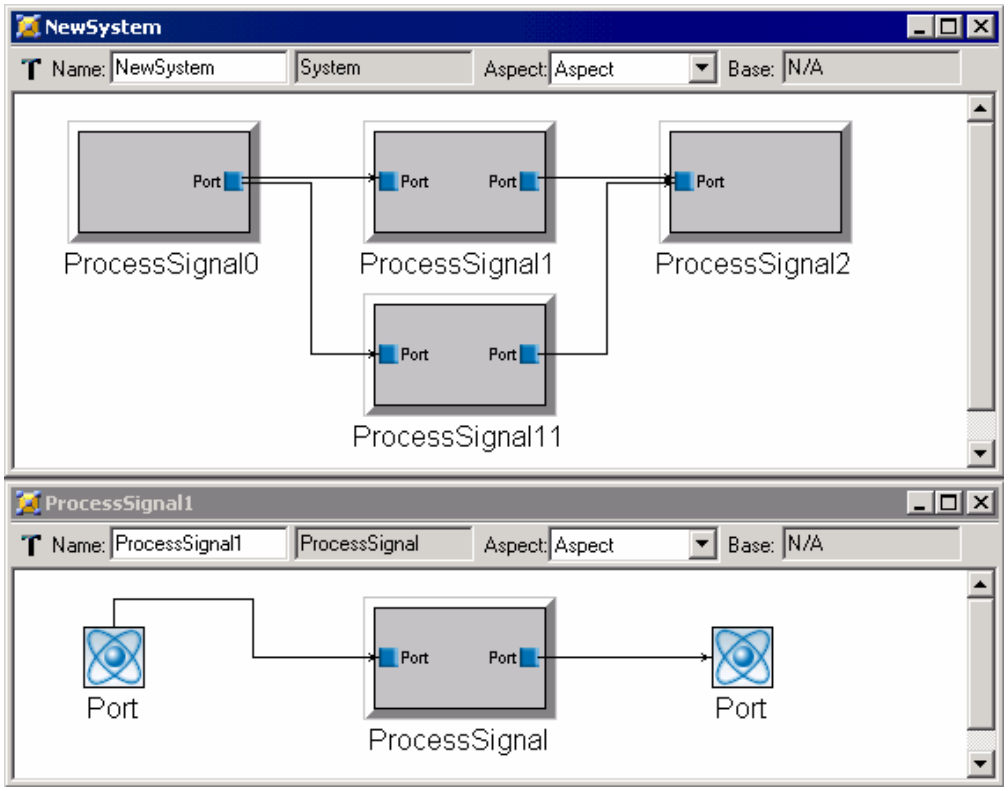


Figure 49. An original domain model in the SignalFlow domain

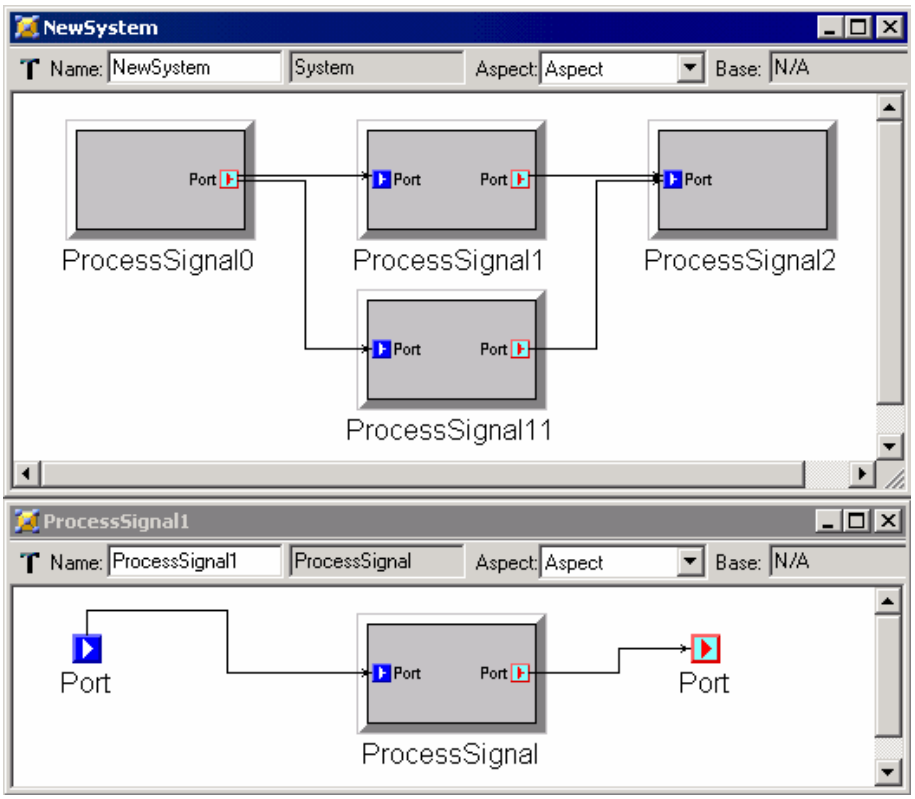


Figure 50. The evolved domain model in the evolved SignalFlow domain

After processing the models in Figure 49 with the domain evolution algorithm created in the earlier part of the section, the output models are ready for use in the evolved domain. Figure 50 shows the evolved domain models. Once again, it is important to note that the icon with blue filling and white arrow represents `Input` type, and white with red arrow and border represents `Output` type (contains a border). The example evolved perfectly, and satisfies all of the syntactic and semantic constraints of the evolved domain.

Evolution through Removal of Type

Recall that domain evolution requires modifications to the domain models when changes occur to any portion of the paradigm tuple (syntax, static semantics, ontology,

and semantic translator). The previous example was a change in all four portions of the tuple, because the ontology set increased. This example is a change in syntax due the *decrease* in the size of the ontology set. Whereas the previous example was required because of changes to the domain (Ports were being eliminated) this example is driven by changes to the language only – that is, the abstraction of the system is changing, but the system itself is not. This is an example of model migration being required by a change to the modeling environment for the sake of convenience, rather than an evolution of the domain concepts.

The Embedded Systems Modeling Language (ESML), a domain-specific graphical modeling language developed for modeling Real-Time Mission Computing Embedded Avionics applications. The following description of the ESML language is taken from [102].

The ESML is specifically intended for modeling component-based avionics applications designed for the Bold-Stroke [103] component deployment and distribution middleware infrastructure. The infrastructure implements an event-driven model of computation. In this model of computation supplier components notify other component of the availability of data through events. Consumer components interested in the data of the suppliers get notified when they subscribe to these events. An event-channel service as implemented by the infrastructure facilitates event filtering, propagation and notification. The real-time nature of event service allows managing and preserving component and task priorities. Details of this infrastructure can be seen in [104].

Because of the complexity of the models created by hand in Bold-Stroke this domain-specific modeling environment proved to streamline the creation of models.

However, some techniques used to abstract the domain concepts into the modeling language were difficult to use, and presented usage issues both by the end user as well as by the developer (Vanderbilt University). Thus a change to the language was proposed in order to further enhance the usability of the modeling environment.

The existing paradigm, which will be referred to as ESML, used the notion of unary association to model Bold-Stroke objects. This unary association (a Reference type named `ComponentProxy` in the GME metamodel) refers to another component type in the Bold-Stroke document. The new paradigm, which will be referred to as ESML', uses a GME run-time feature called *types-and-instances* to model this relationship by creating an instance (at runtime) of the object that was previously referred. In GME an instance is required to have the same structure as its type, but the instance is allowed to modify its attribute values (and associations in which the instance takes part) to vary instance-by-instance. For more information on types-and-instance GME, please refer to the GME user manual [100].

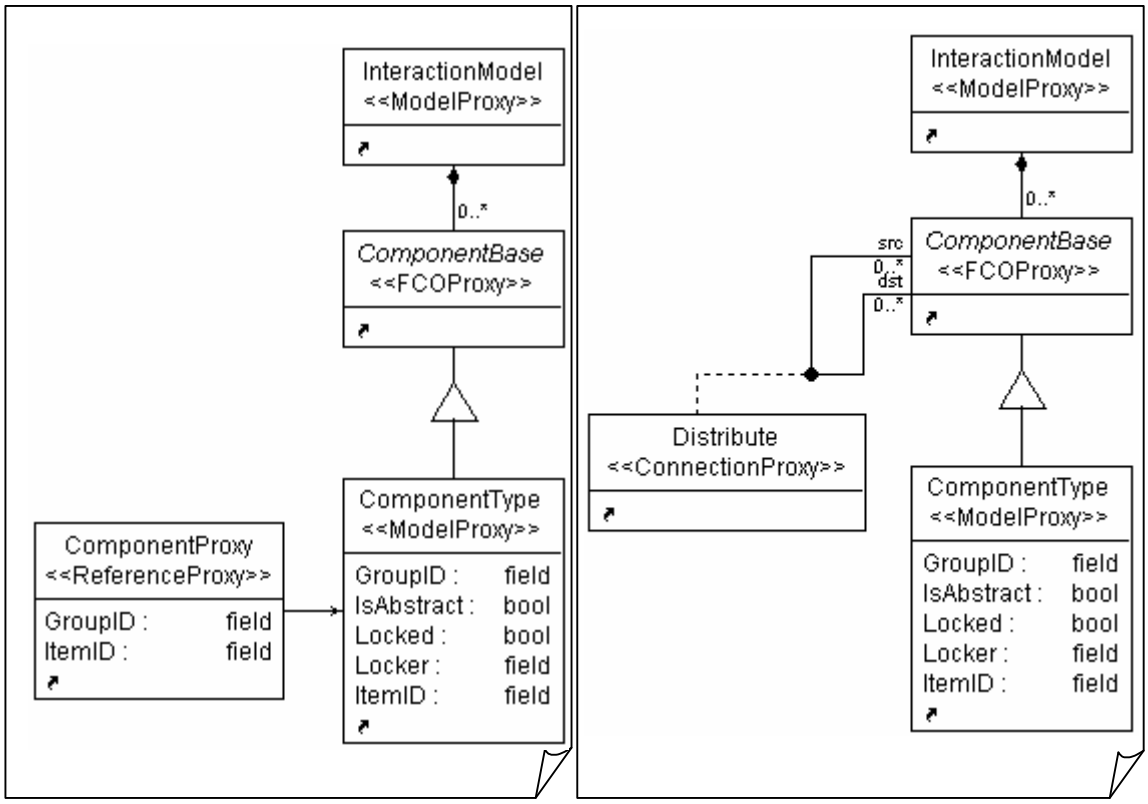


Figure 51. Excerpts from the existing ESML metamodel and the evolved ESML' metamodel

Figure 51 shows the important excerpts from the original and evolved ESML metamodels. The original metamodel contained a `ComponentProxy` class that refers to the `ComponentType` class (a kind of the abstract class `ComponentBase`). In the evolved ESML paradigm, any `ComponentProxy` object should become instead a `ComponentType` object that is a GME-instance of the `ComponentType` object to which the `ComponentProxy` object originally referred. In addition, a connection that exists between an existing `EventType` and the `PublishPort` of the existing `ComponentType` should be duplicated for a new `EventType` instance and the newly instantiated `ComponentType`. Finally, a connection of type `Distribute` should be created that links the `ComponentType` archetype to the `ComponentType` instance.

Part of the reason for including this wordy definition is to underscore the inadequacy of the English language definition of model migration algorithms. However, the description is a fairly accurate portrayal of the diagram shown in Figure 52. The left-hand side of the figure is filled with patterns from the existing metamodel (shown in blue on the diagram) and the right-hand side is filled with consequences of that matched pattern (shown in red).

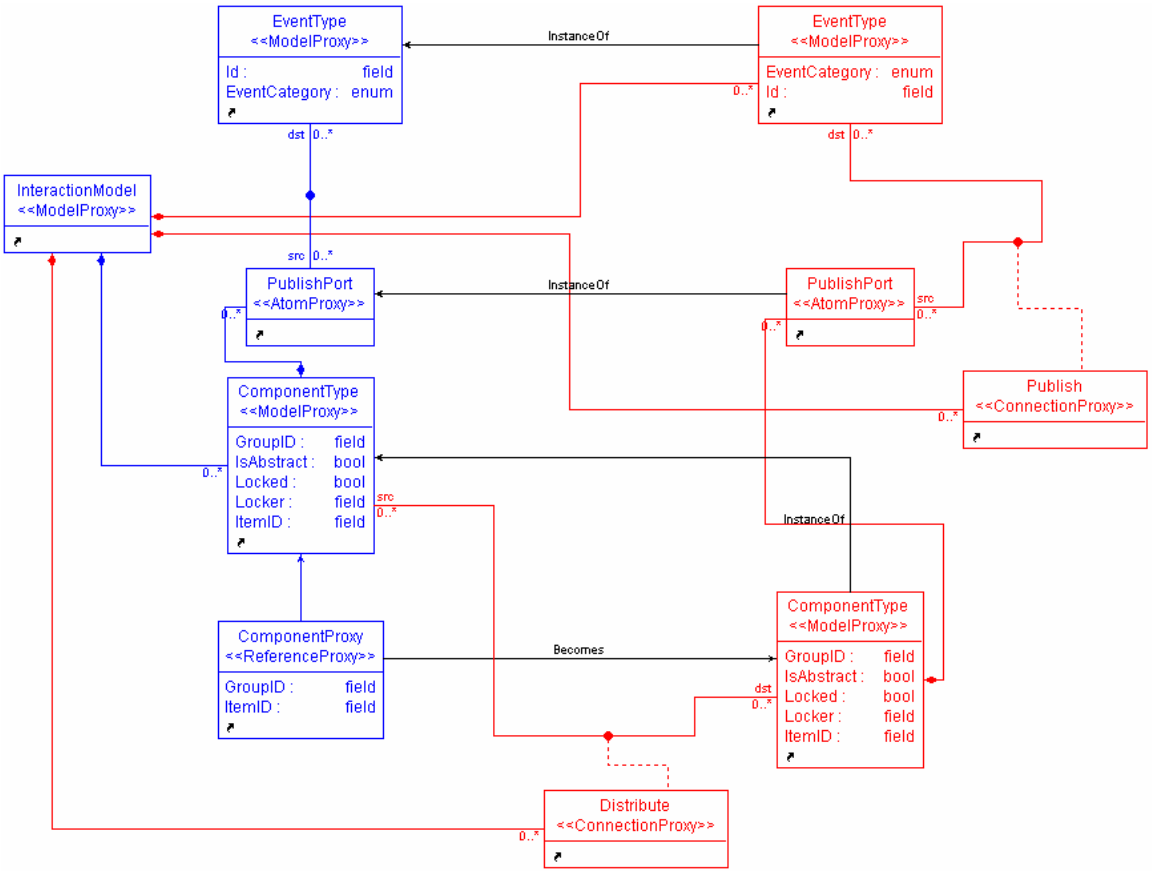


Figure 52. Rule to migrate the ComponentProxy to ComponentType

This figure utilizes nearly every concept in the model migration language for GME. It also exemplifies the difference in the model of computation of the domain evolution framework versus that of the graph-rewriting language used (XSL).

Specifically, the `InteractionModel` on the left side of the diagram serves as a part of the pattern match, as well as the parent for several objects created in the aftermath of matching the pattern. Since XSL prescribes that the input graph is not modified, this would clearly be an illegal statement in XSL. However, the mapping of the model of computation onto XSL takes care of this through low-level implementation details, as explained in the previous chapter.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

The domain of domain evolution has long been approached through the application of low-level domain-independent languages to manage the very specific recreation of domain model databases such that those databases are correct in the evolved domain. This dissertation has presented a domain evolution framework that provides a domain-specific solution to the modeler that is specialized on two axes – the meta-metamodel, and the graph-rewriting language.

The domain evolution framework exists independent of these two axes, and has its own model of computation and semantic interpretation of the domain model evolution algorithms. As a domain-specific solution, it provides the ability to specify the migration of domain models using patterns of domain concepts that are mapped to patterns of evolved domain concepts through mappings that are domain evolution concepts. By definition the domain evolution framework operates based on the *difference* between two metamodels, meaning that it satisfies the golden rule of maintenance; only now we are dealing with the maintenance of domains rather than the maintenance of executable code. In addition, the common functionality and user interface of any domain evolution tool are abstracted into this common framework, thus allowing code reuse and model reuse for the domain of domain evolution.

Once specialized on these the meta-metamodel and graph-rewriting axes the domain evolution tool exists as a standalone modeling environment that a metamodeling expert can use to migrate the domain models of *any* evolved metamodel created with this

meta-metamodel: in essence, low-level domain model migration scripts can be replaced with scripts that are generated from graphical, intuitive, metamodel-specific domain model migration specifications.

Comments on Usage

Users of the model migration language for GME generally agreed that its intuitiveness allowed them to migrate their models in terms of concepts they readily understood. In specific, many users have required special attention during the beginning phases of language use, but once they do some examples, they are comfortable with the language.

Remarks on Limitations

The modeling language is customized for the domain of the specification of the migration of domain models due to *evolution of the modeling language*. However small this application area may be, the language may still be used to operate on problems that may be better classified as another type of transform, for example, tool integration. It should be noted that although this language may be used to solve these problems, it may not be practical to do so when compared with another domain-specific language geared toward that particular problem.

The limitation of this language is not in the problem that it can solve, but in the ability of the modeler to conceive the algorithm that will translate models between two domains. Examples such as those presented in this dissertation are consistent with the expected usage of the language, where evolutionary changes have occurred, but

revolutionary changes, such as translation of a UML class diagram to Microsoft PowerPoint picture, could be performed *if the modeler could conceive of the algorithm*.

This particular language is therefore somewhat limited in its application scope, as it is so domain-specific. However, the concepts used to develop the types of transforms used in this framework, as well as the theories behind the semantic model migration and translation algorithms developed with the model and modeler intents in mind, can be used throughout the model transformation (or graph-rewriting) domain.

As far as GME is concerned, the language is limited in that it can operate only on models that are stored in the XML persistency format. It would be an interesting project to implement a migration engine to operated on the MGA binary files that GME stores by default, but the work involved in undertaking this problem vs. the reward renders the possibility minute – especially considering that all GME projects can be saved in XML format, and that previous GME paradigms can be used to open the old binary files.

One important limitation is the inability of the language to check for correctness of the transformed domain models. The DET for GME/XSL does not perform any checks to ensure the correctness of the migrated models, for syntactic, static semantic, or dynamic semantic reasons. Future work in this area could lead to alerts given to the user when consequence objects are created that violate the evolved metamodel. Due to the onus placed on the user to create valid transforms, however, these alerts would function more as a measure to prevent inadvertent errors than an addition tool used to design the model migration specification.

Continuing Research

The modeling language is intuitive when examined after creation, but many of the concepts of the domain which are taken from procedural or functional programming (e.g., `Tests`, `Rules`, `Ports`, etc.) have semantics that may not be intuitively obvious to a new user of the language. The development of more precise static semantics of the language will enable the creation of constraints that will allow users to be more confident of the models they create at design time, and illegal construction can be pointed out before interpretation time.

Other Meta-Metamodels and Graph-Rewriting Engines

Also, the extension of the framework to other meta-metamodels and graph-rewriting languages is ongoing. An example other meta-metamodel is the UDM metamodel, which has a restricted subset of the GME metamodeling types. This tool will likely *not* use XSL as its graph-rewriting language, as the intermediate XML files will be required to utilize a schema to which they will not comply once transformations have begun to take place (this is because the schema for these XML files are DTDs rather than configuration files for the modeling environment, see [32]).

The addition of the GReAT engine as a possible output language would benefit users familiar with the GReAT language. Since this language allows for the traversal and modification of UDM objects – and UDM objects can be created from GME objects using UDM techniques – this is a reasonable way in which to migrate models. Benefits to the generation of this language include graphical debuggers, as well as (possibly) faster execution of the graph transformations.

Framework Enhancements

The domain evolution framework is acceptable as it is currently implemented, but there are several enhancements that can be made, or are currently being made, to increase the usability of the language and make it easier for the creation of model migration algorithms.

Guard Conditions for Sequence Traversal

One ongoing implementation enhancement is the addition of guard conditions to `Sequence` associations. This would allow for a control flow "shorthand" that would prevent the need of a `Test`/`Case` pair to modify the flow of execution in simple situations. `Sequence` type associations would then be permitted to associate with the same `AreSequenced` object on each end (this is to allow for iterative processing of an `AreSequenced` object). Determination of the `Sequence` association to traverse in the case of more than one association would be done through this guard condition, specified in OCL [5], which expresses the conditions required for allowing the traversal. The guard may evaluate only to 'true' or 'false', and may not have side effects. The guard is also allowed to directly reference objects found in the `AreSequenced` object *only* (i.e., it is possible to reference the parent or child of an object, but only if that object is found in the `AreSequenced` diagram). If no guard condition evaluates to 'true', then execution of the engine will terminate on this path of control. If multiple guard conditions are true, then the execution will not be deterministic[4]. In order to maintain existing models, if the

[4] Although a rewriting engine may choose to implement non-deterministic behavior in a consistent way, the model of computation for the language prescribes the order of traversal is not to be guaranteed.

object is the source of only one `Sequenced` association, then the guard is not necessary, but if present, must be true for the traversal to proceed.

Port Parameters for Creating Algorithm Libraries

An additional framework enhancement currently being implemented is the addition of parameterized `Transforms` that allow for the creation of libraries of algorithms. In this scenario, all types of `Transforms` are capable of passing parameters to each other. These `Parameters` are the signatures of items that are being transformed by the specification. The purpose of passing parameters between `Transform` objects is to allow the graph-rewriting engine (used to traverse the input graph and produce the output graph) to take advantage of locality during execution. In a functional language the functions that do the operations are passed values on which to operate. In a similar fashion, the graph-rewriting specification is "passed" parameters upon which to match – in order to decrease the amount of time that it takes a pattern matcher to locate a particular pattern.

The parameters are passed using an intermediary abstract interface called a `Port` (of either type `Input` or `Output`). The reason for this interface is to provide templatized rules that can operate on generic `Input` and `Output` parameters, and those parameters are provided through the `Port` layer when assembled in the `Transformation` layout.

Recommendations for Future Work

The application of this technology to solving the problems set forth by the OMG in their MDA framework [17] is possible. This architecture requires the development of model translators, which operate on models in one domain (the platform-independent

modeling domain, PIM) to another domain (the platform-specific modeling domain, PSM). These transformations are specified using an OMG specification language, but revisions are constantly being made to this specification, and a great deal of skepticism as to its feasibility exists.

This dissertation made mention at several points as to the possible improvement of performance of the graph-rewriting specification. The current set of model databases requiring modification can be transformed by the migration script in less time than it takes to model the model migration specification. Thus, it is not a priority at this time to increase the performance. However, this is an interesting research topic, which could be approached either through refactoring of the XSL stylesheets, or exploration of alternative ways to generate the XSL stylesheets once they have been tested and are no longer in need of debugging.

Another interesting problem would be the specification of a self-referencing semantic translator that would produce as its output the semantic translator for the model migration language. This is an interesting academic exercise, and although it has not been attempted, the problem should be solvable given that the source and destination metamodels are well defined. A subset of this research would be the expression of the abstract algorithms and types of the domain evolution framework using a particular instance of the domain evolution tool. However, this may prove to be futile, as the language would then be restricted to the destination graph-rewriting tool, and the source would be restricted to that particular meta-metamodel.

Recommendations for future work in the GME-XSL migration environment revolve around the idiosyncrasies of the XML storage format. Additional input from the

user could possibly manage the evolution of different *versions* of the XML model databases (since the paradigm information is encoded in these) to ensure that versions not specified by the migration algorithm are not used. This would require changes to the modeling paradigm to encode (along with the class diagrams) the global unique identifier (GUID) of the paradigm with the migration algorithm. Once again, this is a small payoff when compared to the relatively large amount of work, but could be useful if multiple versions of the paradigm are deployed, and end-to-end migration scripts could be daisy-chained to perform the migration.

APPENDIX A

MODELING

Throughout engineering disciplines, one of the crucial portions of the design process is the creation of a system model. In mechanical, structural, and electrical engineering, this often takes the form of a diagram or layout, where the dynamic or static properties of the system are governed by mathematical equations. In these situations, the design is presented, and the mathematical models are used to prove soundness or correctness of the desired properties of the design. Once the models have fulfilled their purpose, then the construction of the system takes place, governed by the design.

The creation and design of software has long been considered as much art as science. Software developers use a variety of methods (clean room, waterfall, USDP, et al.) to produce the final application, but the design of the system is largely abstract. Often, the original design is modified or abandoned throughout the course of development and maintenance, resulting in an existing system that differs from the documentation (the equivalent of the diagram or layout for a structural system) that describes it. This discrepancy between the documentation and the actual system is common in software systems, and results in difficult maintenance of the deployed system.

A solution to this divergence is to integrate the model of the system with the existence of the system, sometimes referred to as Model-Integrated Computing (MIC). The idea in MIC is to create the executable system directly from the model of the system. The executable model of the system is the lowest level of the four-layer modeling

approach (Level 1 in Figure 53), and the model of the system is in Level 2, and sometimes referred to as a domain model.

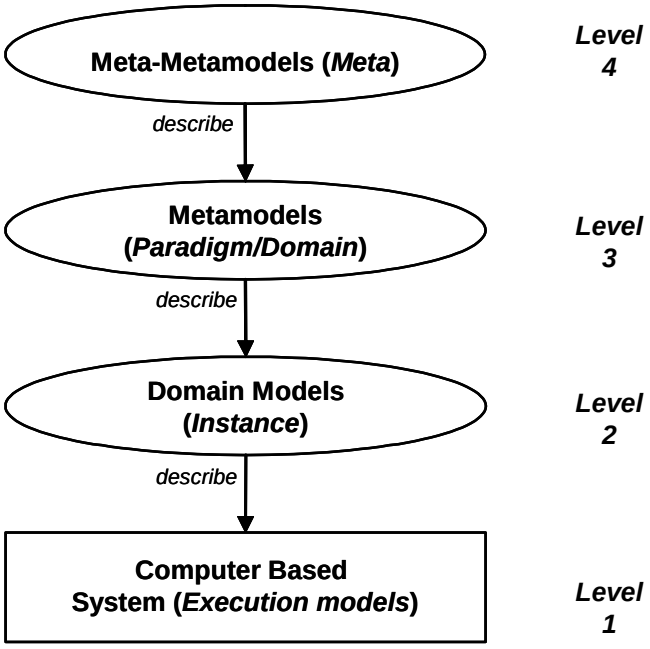


Figure 53. The four-layer metamodeling approach [83]

In order to accurately model the system, a detailed language should exist that can capably and completely describe the behavior of the CBS. This language is called the *paradigm*, and is defined in Level 3, the metamodel level. Using the paradigm defined in Level 3, domain models are defined in Level 2, according to the paradigm language.

The highest level, Level 4, is known as the meta-metamodel layer. The meta-metamodel is used to define a paradigm for metamodeling. In other words, the meta-metamodel models what a metamodel is. It is similar to the definition for the word "definition" in a dictionary.

The benefit of the four-layer modeling approach is its ability to use a metamodeling environment (Level 3) to rapidly develop domain-specific environments

(Level 2) that are capable of producing a CBS (Level 1). The meta-metamodel (Level 4) is the atomic definition of the language that allows metamodels to produce a DSME.

The Parts of a DSME

A DSME exists in two parts: syntax, and semantics [87]. The metamodeling environment provides a way to rapidly design the syntax of a DSME, and its static semantics, which makes up the paradigm. This means that given a paradigm for a domain, it is possible to represent CBSs in that domain using the language of that paradigm. Once these models are created, then they are translated into the CBS execution models through a component of the DSME called the interpreter [3].

The interpreter of a DSME captures the dynamic semantics of the environment in the form of a semantic mapping [86]. The interpreter brings meaning to the models by providing the behavior of the CBS according to each model in the CBS. An interpreter may be seen as a compiler for the language of the paradigm, where the output is not necessarily machine code, but is the required "executable model" that the CBS requires. This output takes the form of computer code, markup language, perhaps even other models. The idea is that the interpreter is the link between the model of the system, and the actual system.

The static semantics of a DSME is generally found in the paradigm in the form of constraints. Constraints assure well-formedness of the models, such that any model may be checked for "correctness" at any time. Constraints do not provide meaning to the models, but instead help to prevent models that are syntactically correct, but semantically ambiguous or incorrect.

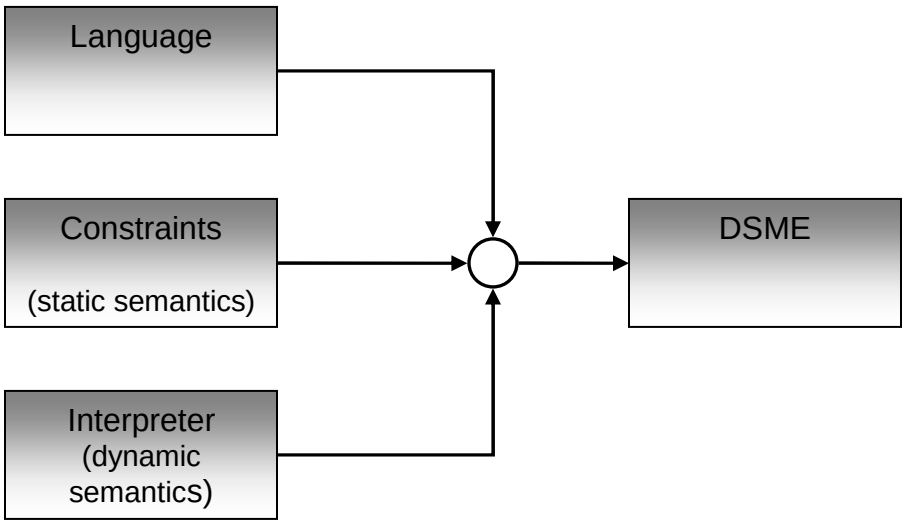


Figure 54. The necessary components of a DSME

The dynamic semantics examines domain models created with this DSME and produces useful domain artifacts that represent (in some form or another) the meaning of the domain models. The dynamic semantics of a DSME is integrally dependent upon the language definition, as it is defined in terms of the language syntax; so any change to the language will require a corresponding change to the interpreter.

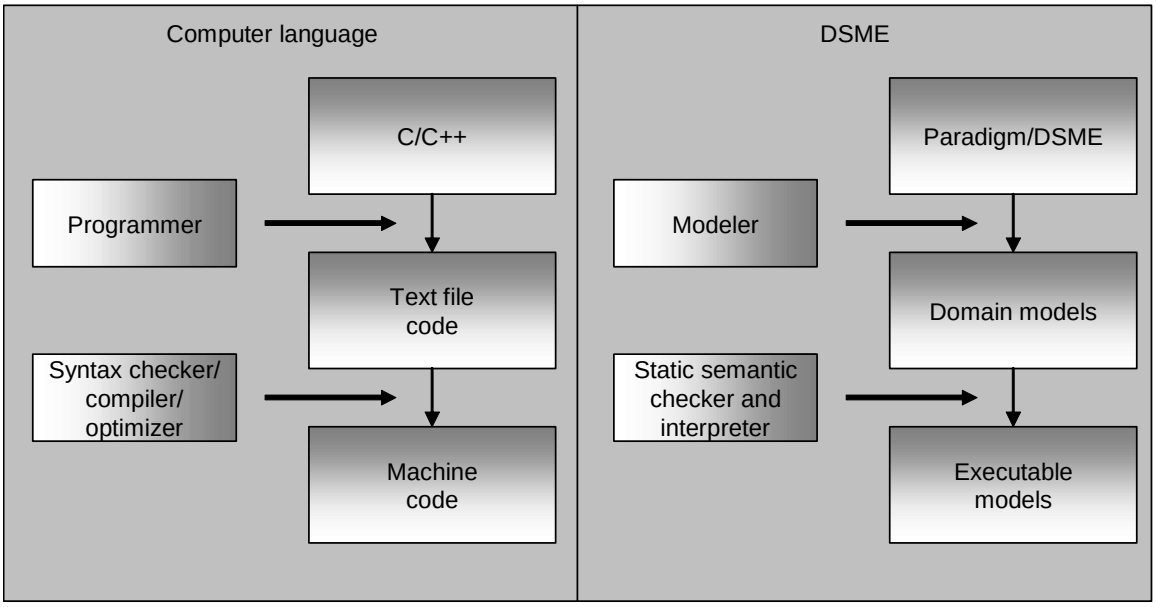


Figure 55. Comparison of the parts of a DSME to those of a traditional programming language

As shown in Figure 54, the combination of the interpreter and paradigm (language and constraints) yields the complete DSME.

The ontology of a DSME may also be described in terms of a textual language such as C/C++. Figure 55 shows the parts of a DSME in context with its corresponding textual language component. Note that the programmer interacts with the languages on the same level, and that some automation transforms the programmer-created components into useful domain artifacts (machine code, or some kind of executable model).

Modeling

The definition of a model varies significantly between fields and researchers in those fields. To an architect, a model is a scale version of a structure or building, used to look for design flaws and possible problems not visible to the mind's eye. To a

mechanical engineer, a model is a mathematical description of some physical system (i.e., a three-bar mechanism, or a heat transfer system) and describes the system to the degree that the solution requires. To a computer scientist, a model is often a class diagram or flowchart, used to describe the structure of a system, or its optimal behavior.

For the purposes of this research, a model can be any of these. All of these types of models (and indeed, all models in general) fall under the broad category of **an abstraction of a system or entity**. Modeling uses models to build up the system (in the abstract world) and examine the system from different aspects according to the purpose of the system (e.g., an architect examines her building from several different angles to ensure proper aesthetics, while a mechanical engineer examines her model for structural integrity, dynamic stability, and energy consumption).

One important feature of modeling CBSs is that an interpreter can transform domain models into executable models that actually run the CBS. Consider the case of an architect. There is no direct application for an architect to build computer-based models of a structural design (that is, an architect's model built from balsa wood has no button to press that will automatically produce a building). However, building a computer-based model of a three dimensional structure allows for analysis of structural and heating/cooling concerns. Even more significant, the system model may be analyzed by a program that can produce custom-scale blueprints from the three dimensional model.

Modeling Languages

Modeling may be simply described as a representation of the entities important in a system. From domain to domain, customized representations could be required in order to fully describe the system. A modeling language is that representation.

To understand how a modeling language exists, first recall the definition of a textual language such as Java. In order to define a class **ClassA** using Java, you use classes, types (int, float, boolean, or classes), inheritance keywords, and libraries of previously created objects. All of these objects used in the definition of **A** are found in Java's list of atomic types (we will call this collection **JavaLang**). When **ClassA** is instantiated into **instance_a**, then the following statements are true,

- instance_a exists.
- **instance_a** has a class definition (metamodel) named **ClassA**.
- **ClassA** has a metamodel named **JavaLang**.
- **instance_a** has a meta-metamodel named **JavaLang**.

In this example, **instance a** is the instance, **ClassA** is the metamodel, and **JavaLang** is the meta-metamodel.

Similar to this textual language, a modeling language is defined in terms of basic concepts that are found in the metamodel. For the UML, these basic concepts are Type, Association, Inheritance, and Constraint [5]. All objects defined for UML tools are defined in terms of these basic concepts. All of the concepts in UML are defined in UML's metamodel, the Meta-Object Facility (MOF) [88]. For a more in-depth comparison of textual and modeling languages, see [89][86].

Once the modeling language is created instance models can be created using the modeling language.

Metamodeling

An interesting implication of modeling domains is that domain modeling is itself a domain [81]. Modeling languages are defined in terms of a meta-metamodel. The definition of a modeling language is not necessarily counter-intuitive, but it is not a trivial process either. However, by modeling a paradigm, then organization concepts such as inheritance, containment, etc., may be used to simplify the creation of the modeling language.

Through a process called metamodeling it is possible to model a modeling language [81]. Visual representation of the metamodeling process allows for the benefits of a visual language such as UML, plus the benefits of modeling the process (e.g., using MIPS to produce artifacts).

The metamodeling domain is designed to produce domain-specific languages. In each case, the generated language is customized for a particular domain. By modeling these languages, and creating a library of defined languages, it is possible to evolve a modeling language, in addition to using previously defined languages in the definition of a new language (e.g., using the UML domain to specify a Java class hierarchy domain).

APPENDIX B

MORPHOLOGICAL NOTATION

The formal specification of morphological operators and functions is instrumental in their terse and unambiguous definition. As with many mathematical definitions, the nomenclature and symbols used are not always intuitive to a reader unfamiliar with that branch of mathematics. This appendix serves as a quick introduction to morphological notation as well as a limited reference for the definitions of operators and symbols used throughout the text. For an in-depth review of these and other basic notations refer to [105] (from which these definitions were taken).

Abstract relationships are sometimes applied to objects that are, to the visual sense, unrelated. An example is two persons, identified as human, who may not be touching each other, but are obviously related if not by family, then at least by their common membership in the human race. These two persons may be grouped into any number of mathematical containers, for instance a tuple, pair, or set. A *set* is an unordered container of objects. A *tuple* is an ordered set, and a *pair* is a tuple with exactly two elements. A *relation* is a set of tuples, all belonging to the same Cartesian product.

Operations on the members of sets are called functions. Specifically, a *function* is a relation that associates a unique element in its range for every element in its domain. That is, each element in its domain maps to a unique element in its range (for more information on mapping, see Appendix C). More generally, this function is known as a total function, because each element of the range set is in the range of the function.

However, there are other types of functions that have a different domain and range than the sets in which they are defined to operate. A full listing of these function classifications, and the notation used to describe them, is given in Table 4.

Table 4. Classification of functions [105]

Function Type	**Description**	**Notation**
Partial	Domain of f is a subset of A	$f: A \nrightarrow B$
Total	Domain of f is equal to A	$f: A \rightarrow B$
Surjective or onto	Range of f is equal to B	$f: A \twoheadrightarrow B$
Injective or one-to-one	Function whose inverse is also a function (1) Partial version (2) Total version	$f: A \rightarrowtail B$ $f: A \rightarrowtail B$
Bijective	Both surjective and injective (i.e., onto and one-to-one)	$f: A \rightarrowtail\!\!\!\!\!\twoheadrightarrow B$
Finite Mappings	Function that is a finite set	$f: A \nrightarrow\!\!\!\!\!\twoheadrightarrow B$

APPENDIX C

MAPPING CONCEPTS

As Harel says in [86], "the degree of formality of a language is independent of its appearance." What the degree of formality *is* dependent on, is how well-defined the semantic mapping of the language is. Given a syntax L, and a semantic domain, S, the semantic mapping, M, is defined as,

$$M : L \rightarrow S$$

The mapping, M, operates on the syntax to produce the semantics. The way in which M is specified is to identify syntax patterns that translate to the appropriate semantic pattern. The form of the semantic pattern is in whatever form deemed appropriate for the semantics (e.g., numbers, letters, sentences, fragments, etc.). Mappings may be specified for an entire domain (as above), or as function operators on individual syntax fragments of a domain, such as below:

$$f : l \rightarrow s(l)$$

Where $s(l)$ is the semantic definition of l according to S. The entire set of functions for a domain, D, makes up the entire mapping of D as shown here:

$$M_D : \{ l \rightarrow s(l) \mid l \in L, s(l) \in S \}$$

As explained in [14] a mapping is a well known concept in mathematics under the names of function, transformation, operator, and morphism (among others). Some examples will enable a better understanding of the map concept, and how it relates to domain-specific modeling in particular. Consider the following mapping:

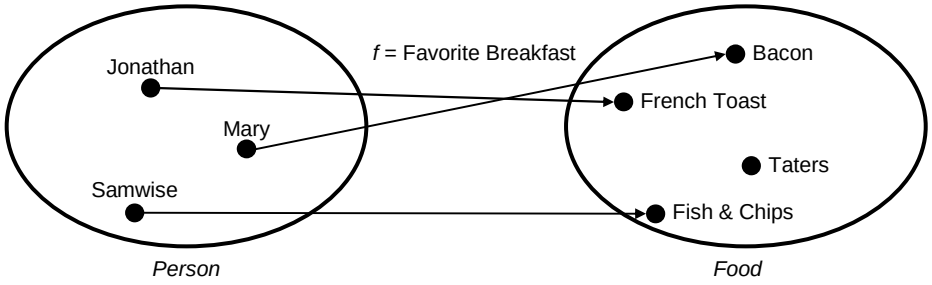


Figure 56. Mapping of a person to favorite breakfast (adapted from [59])

Figure 56 is an example of a mapping named "Favorite Breakfast". In this mapping, the input or "source" objects are people, and the output or "destination" objects are foods. There are some important rules for a mapping, which are related here,

- From each entity in the source domain, there is exactly one arrow leaving the domain
- To any entity in the destination domain, there may be from zero to one or more arrows arriving.

The immediate consequence of these rules are exactly those of the mathematical definition of a well-defined function: that for each value x, there is exactly one value $f(x)$. Also, that for any value, $f(g)$, there may be 1 or more values of x that produce $f(g)$.

Another concept that is familiar from mathematics is the identity function. That is, that for an x, $f(x) = x$. The category map for this function is provided in Figure 57.

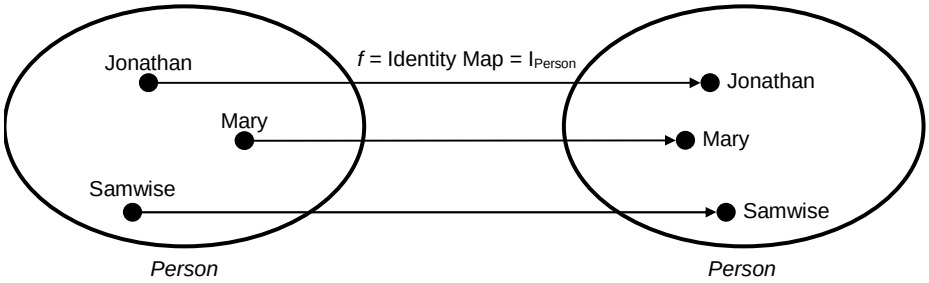


Figure 57. The identity map (adapted from [59])

The identity function serves a useful purpose for the replacement method of graph-rewriting. When used effectively, the identity map may be used to map ideas that are equivalent in the source and destination domains. Then, other individual maps are used to operate on different patterns of the source domain to produce the appropriate destination domain patterns, so that the destination domain semantics are satisfied. The mathematical formalism for this methodology is given below:

$$M_{GR\text{-}Replace}: \quad \left\{ \begin{array}{ll} l \rightarrow f(l) & f(l) \in Transform \\ l \rightarrow I_{src}(l) & f(l) \notin Transform \end{array} \right\}$$

Where *Transform* is the graph transformation as described. In this manner, the transform function is selected when the appropriate syntax is matched, else, this syntax pattern is merely copied using the identity transformation of the source domain.

For a more in-depth discussion of the nuances of category and mapping theory, please refer to [14].

REFERENCES

[1] J. Sztipanovits, G. Karsai, "Model-Integrated Computing", *IEEE Computer*, pp. 110-112, April 1997.

[2] T. Biggerstaff. "Control Localization in Domain Specific Translation" *Proc. 7th International Conference on Software Reuse: Methods, Techniques, and Tools*, LNCS 2319, pp. 153-165, Springer, Berlin, 2002.

[3] J. Sztipanovits, et al., "MULTIGRAPH: An Architecture for Model-Integrated Computing", *Proc. IEEE International Conference on Engineering of Complex Computer Systems*, pp. 361-368, 1995.

[4] J. W. Backus, "The Syntax and Semantics of the Proposed International Algebraic Language of the Zürich ACM-GRAMM conference", *ICIP Paris*, 1959.

[5] OMG Unified Modeling Language Specification, ver. 1.4, Object Management Group, et al., September 2001.

[6] Object Management Group, Object Constraint Language Specification, OMG Document formal/01-9-77. September 2001.

[7] Simonyi, C. "The Future Is Intentional", *IEEE Computer*, Vol. 32, No. 5, pp. 56-57, May 1999.

[8] W. Aitken, et al. "Transformation in Intentional Programming". *Proceedings of the Fifth International Conference on Software Reuse*, pp. 114-123, June 2-5, 1998.

[9] Röder, L. "Transformation and Visualization of Abstractions using the Intentional Programming System", Bauhaus−University, Weimar, Germany.

[10] S. Johnson, "A Portable Compiler: Theory and Practice", *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, pp. 97-104, January 1978.

[11] M. Ganapathi, C. Fischer, "Affix Grammar Driven Code Generation", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 4, pp. 560-599, October 1985.

[12] M. Ganapathi, C. Fischer, J. Hennessy, "Retargetable Compiler Code Generation", *ACM Computing Surveys*, Vol. 14, No. 4, pp. 573-592, December 1982.

[13] Mars Polar Lander website. http://mars.jpl.nasa.gov/msp98/news/mco990930.html

[14] F. W. Lawvere, S. Schanuel, *Conceptual Mathematics: A First Introduction to Categories*, Cambridge University Press, Cambridge, UK, 1997.

[15] D. West, *Introduction to Graph Theory*, 2nd edition. pp. 289-90. Prentice Hall, Upper Saddle River, NJ. 2001.

[16] D. H. Akehurst, "Model Translation: A UML-based specification technique and active implementation approach". Ph. D. Thesis. University of Kent at Canterbury, United Kingdom, December 2000.

[17] J. Bézivin, N. Ploquin, "Tooling the MDA framework: a new software maintenance and evolution scheme proposal", *OOPSLA, 2001: Workshop on Engineering Complex Object-Oriented Systems for Evolution*, Tampa Bay, FL, 2001.

[18] A. Agrawal, T. Levendovszky, J. Sprinkle, F. Shi, G. Karsai, "Generative Programming via Graph Transformations in the Model-Driven Architecture", *OOPSLA - Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, WA, November 2002.

[19] F. Keienburg, A. Rausch, "Using XML/XMI for Tool Supported Evolution of UML Models", *34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, Vol. 9, p. 9064, Maui, HI, January 3-6, 2001.

[20] A. Schürr, "PROGRES for Beginners", Lehrstul für Informatik III, University of Aachen.

[21] D. Blostein, A. Schürr, "Computing with Graphs and Graph-Rewriting", *Software – Practice and Experience*, 6th Proceedings in Informatics, pp. 1-21, 1997.

[22] A. Schürr, "Specification of Graph Translators with Triple Graph Grammars". University of Aachen, AIB 94-12.

[23] M. Nagl, "Set Theoretic Approaches to Graph Grammars", *Proc. 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, Springer-Verlag LNCS 291, pp. 41-54, 1987.

[24] A. Schürr, "Adding Graph Transformation Concepts to UML's Constraint Language OCL", *Electronic Notes in Theoretical Computer Science*, Amsterdam, 2000.

[25] P. Braun, F. Marschall, "BOTL: The Bidirectional Object Oriented Transformation Language". TUM-I0307, Institute for Informatics, Technical University of Munich.

[26] G. Engels, R. Heckel, "From Trees to Graphs: Defining the Semantics of Diagram Languages with Graph Transformation", *ICALP Satellite Workshops*, Proceedings in Informatics, pp. 373-382, 2000.

[27] A. Schürr, "Programmed Graph Replacement Systems". In *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations*, pp. 479-546. World Scientific, Singapore, 1997.

[28] A. Agrawal, G. Karsai, F. Shi, "Interpreter Writing Using Graph Transformations", ISIS Technical Report, TR#ISIS-03-401, 2003.

[29] T. Levendovszky, G. Karsai, A. Ledeczi, M. Maroti, et al., "Model Reuse with Metamodel-Based Transformations", *7th International Conference on Software Reuse: Methods, Techniques, and Tools*, *Lecture Notes in Computer Science 2319*, pp. 166-178, Austin, TX, April 2002.

[30] A. Agrawal, T. Levendovszky, J. Sprinkle, F. Shi, G. Karsai, "Generative Programming via Graph Transformations in the Model-Driven Architecture", *OOPSLA - Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, WA, November 2002.

[31] G. Karsai, A. Agrawal, F. Shi, J. Sprinkle, "On the use of Graph Transformations in the Formal Specification of Computer-Based Systems", *IEEE TC-ECBS and IFIP10.1 Joint Workshop on Formal Specifications of Computer-Based Systems*, pp. 19-27, Huntsville, Alabama, April 2003.

[32] A. Bakay, "The UDM Framework," http://www.isis.vanderbilt.edu/Projects/mobies/

[33] The Extensible Stylesheet Language, http://www.w3.org/Style/XSL/.

[34] "Turing Machine Markup Language", http://www.xml.com/pub/r/1036

[35] B. Lerner, "A Model for Compound Type Changes Encountered in Schema Evolution", *ACM Transactions on Database Systems*, Vol. 25, No. 1, pp. 83-127, March 2000.

[36] N. C. Shu, B. C. Housel, V. Y. Lum, "CONVERT: A High Level Translation Definition Language for Data Conversion", *Communications of the ACM*, Vol. 18, No. 10, pp. 557-567, October 1975.

[37] B. Shneiderman, G. Thomas, "An Architecture for Automatic Relational Database System Conversion", *ACM Transactions on Database Systems*, Vol. 7, No. 2, pp. 235-257, June 1982.

[38] S. B. Navathe, "Schema Analysis for Database Restructuring", *ACM Transactions on Database Systems*, Vol. 5, No. 2, pp. 157-184, June 1980.

[39] Y. Chen, "Integrating Heterogenous OO Schemas", *Journal of Information Science and Engineering*, Vol. 16, No. 3, pp. 555-591, July 2000.

[40] J. Banerjee, et al., "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", *Proceedings of the ACM-SIGMOD Conference on Management of Data*, San Francisco, CA, May 1987.

[41] H. Kim, H. Korth, "Schema versions and DAG rearrangement views in Object-Oriented Databases", Technical Report, TR-88-05, University of Texas at Austin, Austin, TX, 1988.

[42] D. J. Penney and J. Stein, "Class modification in the GemStone object-oriented DBMS", *Proceedings on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA), pp. 111-117, Orlando, FL, October 1987.

[43] A. H. Skarra, S. B. Zdonik, "The Management of Changing Types in an Object-Oriented Database", *Proceedings of OOPSLA '86*, pp. 483-495, Portland, OR, September 1986.

[44] S. M. Clamen, "Schema Evolution and Integration", *Distributed Parallel Databases*, Vol. 2, No. 1, pp. 101-126, January 1994.

[45] S. Bratsberg, "Unified Class Evolution by Object-Oriented Views", *Proceedings of the 11th International Conference on the Entity-RelationshipApproach*, pp. 423-439, October 1992, Karlsruhe, Germany.

[46] S. Monk, I. Sommerville, "A Model for Versioning Classes in Object-Oriented Databases", *Proceedings of the Tenth British National Conference on Databases*, Aberdeen, Scotland, 1992.

[47] M. Tresch, M. H. Scholl, "Meta Object Management System and Its Application to Database Evolution", *Proceedings of the 11th International Conference on the Entity-RelationshipApproach*, pp. 299-321, October 1992, Karlsruhe, Germany.

[48] S. E. Lautemann, "A Propagation Mechanism for Populated Schema Versions", *Proceedings of the International Conference on Data Engineering*, pp. 67-78, Birmingham, UK, April 1997.

[49] S. E. Lautemann, "Schema Versions in Object-Oriented Database Systems", *Proceedings of the 5th International Conference on Database Systems for Advanced Applications*, Melbourne, Australia, April 1997.

[50] Y. G. Ra, E. A. Rundensteiner, "A Transparent Object-Oriented Schema Change Approach Using View Evolution", Technical Report, University of Michigan, Ann Arbor, MI, 1994.

[51] P. Breche, "Advanced Primitives for Changing Schemas of Object Databases", *Proceedings of the 1996 Conference on CaiSE*, Heraklion, Crete, May 1996.

[52] P. Breche, F. Ferrandina, M. Kuklok, "Simulation of Schema Change Using Views", *Proceedings of the 6th International Conference on Database and Expert Systems Applications*, London, UK, September 1995.

[53] F. Ferrandina, S.-E. Lautemann, "An Integrated Approach to Schema Evolution for Object Databases", *Proceedings of the Third International Conference on Object-Oriented Information Systems*, pp. 280-294, December 1996.

[54] R. J. Peters, M. T. Öszu, "An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems", *ACM Transactions on Database Systems*, Vol. 22, No. 1, pp. 75-114, March 1997.

[55] D. Garlan, et al., "TransformGen: Automating the Maintenance of Structure-Oriented Environments", *ACM Transactions on Programming Language Systems*, Vol. 16, No. 3, pp. 727-774, May 1994.

[56] N. Habermann, D. Notkin, "Gandalf: Software Development Environments", *IEEE Transactions on Software Engineering*, Vol. 12, No. 12, pp. 1117-1127, December 1986.

[57] N. Habermann, D. Garlan, D. Notkin, "Generation of Integrated Task-Specific Software Environments", *CMU Computer Science: A 25th Anniversary Commemorative*, R. F. Rashid, Ed. pp. 69-97. ACM Press anthology series. ACM Press, New York, NY. 1991.

[58] B. Lerner, N. Habermann, "Beyond Schema Evolution to Database Reorganization", Proceedings of the Joint ACM European Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOSPLA/ECOOP '90), pp. 67-76, Ottawa, Canada, October 1990.

[59] C. Liu, P. Chrysanthis, S. Chang, "Schema Evolution Through Changes to ER Diagrams", *Journal of Information Science and Engineering*, Vol.9 No.4, pp.657-683, December 1993.

[60] K. Claypool, E. Rundensteiner, G. Heineman, "ROVER: Flexible Yet Consistent Evolution of Relationships", *Data & Knowledge Engineering*, Volume 39, No. 1, pp. 27-50, October 2001.

[61] N. Pittas, A. C. Jones, W. A. Gray, "Evolution Support in Large-Scale Interoperable Systems: a Metadata Driven Approach", *Proceedings of the Australiasian Database Conference*, pp. 161-168, Gold Coast, Queensland, Australia, January 2001.

[62] H.-L. Yang, "Reformulating Semantic Integrity Constraints Precisely", *Journal of Information Science and Engineering*, Vol. 11, No. 4, pp. 513-540, December 1995.

[63] Meta Integration Technology, Inc. http://www.metaintegration.net/

[64] Rational Rose. http://www.rational.org/

[65] Oracle. http://www.oracle.com/

[66] Upgrading Netscape Directory Server 4.x. http://developer.netscape.com/docs/manuals/directory/41/in/upgrade.htm#1115984

[67] W. Wulf, R. Johnsson, C. Weinstock, S. Hobbs, C. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, New York, 1975.

[68] S. Johnson, "A Portable Compiler: Theory and Practice", *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, pp. 97-104, January 1978.

[69] M. Ganapathi, C. Fischer, J. Hennessy, "Retargetable Compiler Code Generation", *ACM Computing Surveys*, Vol. 14, No. 4, pp. 573-592, December 1982.

[70] U. Aßmann, "Graph Rewrite Systems for Program Optimization", *ACM Transactions on Programming Languages and Systems*, Vol. 22, No. 4, pp. 583-637, July 2000.

[71] *CDIF Framework for Modeling and Extensibility*, Extract of Interim Standard EIA/IS-107, Electronics Industries Association, CDIF Technical Committee, January 1994.

[72] J. Gray, B. Ryan, "Applying the CDIF Standard in the Construction of CASE Design Tools", *Australian Software Engineering Conference*, p. 88, Sydney, Australia, September 1997.

[73] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, H. Zheng, "HyVisual: A Hybrid System Visual Modeler", Technical Memorandum UCB/ERL M03/1, University of California, Berkeley, January 28, 2003.

[74] Y. Hur, I. Lee, "Distributed Simulation of Multi-Agent Hybrid Systems", *IEEE International Symposium on Object-Oriented Real-time distributed Computing*, April 29-May 1, 2002.

[75] B. I. Silva, K. Richeson, B. H. Krogh, A. Chutinan. "Modeling and verification of hybrid dynamical system using CheckMate", *ADPM 2000*, September 2000.

[76] HSIF: Hybrid Systems Interchange Format, Vanderbilt University. http://micc.isis.vanderbilt.edu:8080/HSIF

[77] R. Lemesle, "Transformation Rules Based on Meta-modeling", Proceedings of the Second International Enterprise Distributed Object Computing Workshop, San Diego, November 1998.

[78] D. Milićev, "Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments", *IEEE Transactions on Software Engineering*, pp. 413-431, April 2002.

[79] D. Milićev, "Domain Mapping Using Extended UML Object Diagrams", *IEEE Software Engineering*, pp. 90-97, March/April 2002.

[80] C. Thomason IV, M.S. Thesis, Vanderbilt University, Nashville, TN. May 2000.

[81] G. Nordstrom, J. Sztipanovits, G. Karsai, A. Ledeczi, "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments", *Proceedings of the IEEE ECBS'99 Conference*, pp. 68-74, Nashville, Tennessee, April, 1999.

[82] G. Nordstrom, J. Sztipanovits, G. Karsai, "Metalevel Extension of the MultiGraph Architecture", *Proceedings of the IEEE ECBS'98 Conference*, pp. 61-68, Jerusalem, Israel, April 1998.

[83] J. Sprinkle, G. Karsai, A. Ledeczi, G. Nordstrom, "The New Metamodeling Generation", *Proceedings of the IEEE ECBS '01 Conference*, Washington, D.C., 2001.

[84] The Domain Modeling Environment (DoME). http://www.htc.honeywell.com/dome/index.htm

[85] MetaEdit+, Domain specific modeling tool. http://www.metacase.com/

[86] D. Harel, B. Rumpe, "Modeling Languages: Syntax, Semantics, and All That Stuff. Part I: The Basic Stuff", Technical Report MCS00-16, Mathematics & Computer Science, Weizmann Institute Of Science, Rehovot, Israel.

[87] G. Karsai, G. Nordstrom, A. Ledeczi, J. Sztipanovits, "Specifying Graphical Modeling Systems Using Constraint-based Metamodels", *Proceeding of the IEEE Symposium on Computer Aided Control System Design*, Anchorage, AK, 2000.

[88] Meta Object Facility (MOF) Specification, ver. 1.4, Object Management Group, et al., April 2002.

[89] D. Batory, D. McAllester, L. Coglianese, W. Tracz, "Domain Modeling in Engineering Computer-Based Systems", *Proceedings of the 1995 International Symposium and Workshop on Systems Engineering of Computer Based Systems*, pp. 19-26, 1995.

[90] J. Parsons, Y. Wand, "Emancipating Instances from the Tyranny of Classes in Information Modeling", *ACM Transactions on Database Systems*, Vol. 25, No. 2, pp. 228-268, June 2000.

[91] D. Harel, "StateCharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, Vol. 8, pp. 231-274, 1987.

[92] D. Harel, A. Naamad, "The STATEMATE Semantics of Statecharts", *ACM Transactions on Software Engineering Methods*, Vol. 5, No. 4, 1996.

[93] The Worldwide Web Consortium, The Extensible Stylesheet Language (XSL), http://www.w3.org/Style/XSL/.

[94] M. Kay, *XSLT – Programmers Reference*, Wrox Press Ltd., 2nd Edition, 2002.

[95] The Worldwide Web Consortium, XML Path Language (XPath), http://www.w3.org/TR/xpath

[96] G. Engels, R. Heckel, From Trees to Graphs: Defining the Semantics of Diagram Languages with Graph Transformation, ICALP Satellite Workshops, Proceedings in Informatics, pp. 373-382, 2000.

[97] A. Schürr, Specification of Graph Translators with Triple Graph Grammars, in: G. Tinhofer, ed., Proc. WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, Herrsching, Germany, LNCS 903, Springer, Berlin, pp. 151-163, 1994.

[98] A. Schürr, Programmed Graph Replacement Systems, in: Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations, World Scientific, Singapore, pp. 479-546, 1997.

[99] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[100] Institute for Software Integrated Systems, The Generic Modeling Environment (GME) http://www.isis.vanderbilt.edu/projects/gme.

[101] A. Ledeczi, et al., "Composing Domain-Specific Design Environments", *IEEE Computer*, Vol., 34, No.11, pp. 44-51, 2001.

[102] S. Neema, A. Bakay, G. Karsai, "Embedded Systems Modeling Language", Institute of Software Integrated Systems, Vanderbilt University.

[103] http://www.rl.af.mil/tech/programs/MoBIES/Boeing.html

[104] http://www.omg.org/news/meetings/realtime2001/abstracts/BoeingAbstract.pdf

[105] N. Nissanke, *Introductory Logic and Sets for Computer Scientists*, Addison Wesley Longman, Harlow, England, 1999.

ELECTRICAL ENGINEERING

METAMODEL BASED MODEL MIGRATION

JONATHAN SPRINKLE

Dissertation under the direction of Dr. Gabor Karsai

Model-integrated computing (MIC) is a methodology for the development of computer-based systems (CBSs). MIC predicates that models are developed according to a particular metamodel (that is, a language representation of the CBS's actual domain). MIC is the integration of models with a way to extract a "meaning" from those models. Using metamodeling, it is possible to model the domain of the CBS and generate a domain-specific modeling environment (DSME) whose ontology is the types of components in the CBS. With this DSME, models of the CBS are constructed and expressed with the ontology of the domain, and then interpreted to produce an executable model.

The true value of domain-specific modeling is found not in the DSME, but the models that are created in that DSME. Changes to the physical system can be modeled, and the resulting executable model then is a working version of the CBS. Unfortunately, if the model of the domain – or metamodel – is changed, all models that were defined using that metamodel may require maintenance to have the semantics that represent the CBS correctly. Without ensuring the correctness of the domain models after a change to the domain, the true value of the DSME will be lost. The only way to use instance

models based on the original metamodel is to migrate them for use in the modified metamodel.

This problem is known as the Model Migration (MM) problem. Current state of the art for Model Migration heavily depends on low-level implementation, and treats the domain models like data files rather than domain-specific models. There are benefits to performing model migration using domain-specific concepts such as mapping objects from one domain to another, and leveraging the existing metamodel structure for designing these mappings. This dissertation extends the state of the art by implementing such a metamodel based model migration language, which is a modeling language in its own right.

Furthermore, a model migration language is dependent on two components: the meta-metamodel, and the graph-rewriting language used to perform the actual translation of the domain models. These parameterizations can be abstracted to reveal a framework that any meta-metamodel and graph-rewriting language would use to perform model migration. This dissertation gives form and function to such a framework, and describes how it can be specialized to yield a particular instance of a framework: a domain evolution tool.

Approved_____ Date_____