COMPUTER SCIENCE

GME-MOF: AN MDA METAMODELING ENVIRONMENT FOR GME

MATTHEW JOEL EMERSON

Thesis under the direction of Professor Janos Sztipanovits

Versatile model-based design demands languages and tools which are suitable for the creation, manipulation, transformation, and composition of domain-specific modeling languages and domain models. The Meta Object Facility (MOF) forms the cornerstone of the OMG's Model Driven Architecture (MDA) as the standard metamodeling language for the specification of domain-specific languages. This thesis describes an implementation of MOF v1.4 as an alternative metamodeling language for the Generic Modeling Environment (GME), the flagship tool of Model Integrated Computing (MIC). This implementation utilizes model-to-model transformations specified with the Graph Rewriting and Transformation tool suite (GReAT) to translate between MOF and the UML-based GME metamodeling language. The technique described by this paper illustrates the role formally well-defined metamodeling and metamodel-based model transformation approaches can play in interfacing MIC technology to new and evolving modeling standards.

Approved ______________________________ Date __________________

GME-MOF: AN MDA METAMODELING ENVIRONMENT FOR GME

By

Matthew Joel Emerson

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Computer Science

May, 2005

Nashville, Tennessee

Approved:	Date:
Janos Sztipanovits	2-28-05
Gabor Karsai	2-28-05

To Deidre.

ACKNOWLEDGEMENTS

This research was sponsored by the National Science Foundation through the Information Technology Research project, under contract number CCR-0225610.

I am grateful to all those who helped me complete this work, most especially my advisor Dr. Janos Sztipanovits. He didn't just provide the vision which guided my work — he also opened my eyes to the importance of this entire area of research. I am further grateful for the cooperation of Kai Chen, Andrew Dixon, Gabor Madl, and Ethan Jackson. Their help, criticism, and advice saw me through a number of rough spots.

I am indebted to many in (and formerly in) ISIS, including Dr. Aditya Agrawal, Endre Magyari, Feng Shi, Attila Vizhanyo, Peter Volgyesi, and Brian Williams, for technical assistance and help. Without the tireless work of the entire GReAT team, my research would have been many times more difficult. Special thanks to Dr. James "Bubba" Davis and Dr. Greg Nordstrom for providing me with my introduction to MIC and guiding some of my initial work. Thanks also to Dr. Gabor Karsai for helping me improve this document.

Finally, thanks to those who have kept my spirits high while I completed this research (and back into hallowed antiquity). Thanks to my parents, Joel and Beth, who always had a home-cooked meal ready when I needed it. Thanks to my sister, Leah, whose life makes mine seem simple by comparison. Finally and most happily, thanks to the lovely Deidre Marie Vaughn, who makes all my labor worthwhile.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

A computer-based system (CBS) is essentially an *integrated* system composed of a computational component (with embedded software), a physical environment, and a sensing and actuation hardware mechanism which establishes an interface between the two. The physical aspects of a CBS, its hardware and environment, impose constraints or requirements on its embedded software. The role of embedded software then is to configure and control the computational components of a CBS to meet these physical requirements. Designing such systems is an inherently complex task, because the constraints posed by the physical aspects of a CBS cross-cut the entire development process - they generally cannot be satisfied through a single design decision. To cope with this complexity, developers must turn to model-based design techniques, which can holistically addresses the many interdependent physical, functional and logical aspects of CBS design.

The advent of Model Driven Architecture (MDA) marks the beginning of the embrace of model-based software design techniques by mainstream software developers. As the central vision of the OMG, MDA proposes the specification of software systems through modeling and model transformation[15][7]. It advocates the development of domain-specific software applications through modeling to capture software requirements and design, platform, and deployment specifications. Generally, MDA limits the role of model transformations to one-shot mappings from abstract platform-independent models (or PIMs) to implementable platform-specific models (or PSMs). One theme of MDA is that OMG's widely-successful Universal Modeling language (UML), which provides a common graphical syntax for object-oriented design[16], will be the single, universal, platform-independent modeling language used by model translators to generate software artifacts for specific platforms. The basis of this conviction stems from viewing model-based design in the same light as conventional programming, where language standardization has been an important issue. However, the scope of model-based design is in fact much broader. Model-based design encompasses the entire modeling process, which inherently includes the selection of essential domain aspects, careful separation of the modeled and not modeled worlds, and abstraction. UML alone cannot sufficiently model all software applications, let alone inherently more complicated computer-based systems, primarily due to its limited scope and flexibility. UML lacks native facilities for describing specialized software domains such as distributed real-time systems, and it provides no way to model the physical properties of embedded systems.

In an effort to provide a mechanism for expanding its limited scope, UML evolved from a single monolithic language into a familty of closely-related languages which all extend a common UML core. These languages are known as UML Profiles[16]. UML Profiles are stereotyped packages that contain model elements extended with stereotypes, tagged values and constraints. Unfortunately, profiling is not powerful enough to change the fundamental syntactic and semantic properties of UML because it merely constrains existing UML constructs rather then modifying or add new ones. Profiling also tends to create a complex web of interfering standards as different domain modelers profile UML in different ways to capture the same domain's concepts.

The realization of the insufficiencies of UML gives rise to the question, what *is* the right way to model CBSs? Which modeling language should we use? Modeling languages designed to capture the interesting properties of software systems, such as UML, generally lack the necessary facilities for modeling entire CBSs. The models must also capture the physical properties of the platforms and the embedding environment to make these properties computable and analyzable. While UML includes some diagram types useful for modeling dynamic, reactive systems (for example, StateCharts), it is inadequate for capturing models with a continuous-time semantics (for example, systems of ordinary differential equations). Furthermore, the scope of modeling and the level of abstraction required for designing CBSs are highly domain-specific. We cannot expect that the same kinds of models and modeling languages which may be effectively used to design controllers for brake-by-wire systems in cars (where safety, timing and cost are the critical properties) may be used in designing mobile phones (where cost, power, security, and feature richness are the most important factors). Finally, mature application areas, including engineering disciplines such as control theory or mechanical engineering, generally have their own equally-mature domain-specific terminology and concepts, and forcing the use of another unrelated set of concepts for modeling within such a domain seems both awkward and wasteful. Consequently, it seems that *that there is no single, universal modeling language capable of satisfying the requirements of all CBSs*[12].

Designing computer-based systems requires the use of models based on domain knowledge and terminology. This requires the invention of many modeling languages, each specific to an application domain. The more radical approach of constructing domain-specific modeling languages (DSMLs) demands an understanding of the fundamentals of constructing modeling languages and creating standards and tool suites for facilitating their specification and composition. Model-Integrated Computing (MIC) is a domain-specific, model-driven approach to system development which uses models and model transformations as first-class artifacts, and where every model is a valid statement from some DSML. MIC captures the core characteristics of a domain in the fixed constructs of a DSML and captures the variability of a domain through the domain models[11][26]. Over the last ten years, MIC metamodeling approaches have been successfully applied in a variety of application domains[9][35]. Like MDA, MIC views the model development process as a series of transformations among models — in fact, MIC may be considered a practical manifestation of the MDA vision. However, the primary different between MIC and MDA remains the role of DSMLs.

The vision of the domain-specific approach is that *only* those things important in the domain are available to the domain modeler, and its primary supporting artifact is the metamodel. A metamodel is a model of a DSML expressed using some metamodeling language. Metamodeling provides a uniform way to define new modeling languages. The latest developments in UML 2[21] depend on this approach, as the UML 2 family of modeling languages has been defined using the Meta Object Facility (MOF). MOF has emerged as the OMG's standard metamodeling language, and one of the more underutilized MOF use-cases is the specification of DSMLs which are not part of the standard MDA suite of languages[17]. In the future, MOF may serve as a widely-adopted tool-independent metamodeling language, allowing model data to be freely transferred between compliant tools using OMG's XML Metadata Interchange (XMI) technology[22].

We must also consider the need for powerful tool suites which support model-based design by aiding in specifying, manipulating, transforming, and composing models. With proper tool support, metamodeling allows DSMLs to be created and maintained quickly and inexpensively. The primary MIC development tool is the Generic Modeling Environment (GME)[2], a metaprogrammable model builder for designing and modeling in domain-specific modeling environments. The best method for the rapid creation of domain-specific environments is to create a metamodeling environment used specifically to design them. GME supports its own metamodeling environment and language based on UML class diagrams with class stereotypes and OCL constraints called MetaGME.

Problem Statement

As MOF becomes the widely-adopted, industry-standard metamodeling language, GME must evolve through metaprogramming and model transformation to support tool-independent MOF-based metamodels while also maintaining compatibility with technologies based on its own tool-specific metamodeling language.

This thesis describes the implementation of a MOF v1.4-based[17] alternative metamodeling environment for GME through metamodeling and model-to-model transformation. The transformation allows the new MOF metamodeling environment to leverage existing tool support for GME modeling environment generation. The implementation of the MOF sits as an additional layer of abstraction above the existing GME-specific metamodeling facilities. This work also provides an opportunity to evaluate MOF as a metamodeling language, particularly in terms of its support for DSML composition. The approach taken in this thesis demonstrates the power and flexibility granted by metaprogrammable tool architectures. The primary results of the thesis have been previously described in other academic publications[32][31].

Modeling and Composition of DSMLs

Formally, a DSML is a five-tuple of abstract syntax (A), concrete syntax (C), syntactic mapping (M_C), semantic domain (S), and semantic mappings (M_S) [38]:

$$L = < A,\ C,\ M_C,\ S,\ M_S >$$

DSML syntax is defined in three parts: abstract syntax, concrete syntax, and syntactic mapping. The abstract syntax A defines the language concepts, relationships, and any integrity constraints which restrict the set of well-formed statements from the language. The concrete syntax (C) defines the specific graphical, textual, or mixed notations used to depict model elements. The syntactic mapping $M_C : A \rightarrow C$ assigns syntactic constructs (graphical, textual or both) to the elements of the abstract syntax. DSML semantics consist of two parts: semantic domain and semantic mapping. The semantic domain S is usually defined by means of some mathematical formalism in terms of which the meaning of well-formed domain models is specified. The semantic mapping $M_S : A \rightarrow S$ maps concepts from the abstract syntax to those of the semantic domain.

DSML syntax provides the modeling constructs which conceptually form an interface to the semantic domain through the semantic mapping. Any DSML which is to be used in the development of embedded systems requires the precise, explicit, and complete specification (or modeling) of all five components of the language definition. The languages which are used for defining components of DSMLs through modeling are called metamodeling languages and the modeled, formal specifications of DSMLs are called metamodels[15].

The specification of the abstract syntax of DSMLs requires at minimum a metamodeling language which can model concepts, relationships, and integrity constraints. In GME, UML Class Diagrams and the Object Constraint Language (OCL) form the metamodeling language. This selection is consistent with UML's and MOF's four-layer metamodeling architecture[17]. The elements of the abstract syntax find their representations among the elements of the concrete syntax through the syntactic mapping. The mechanism for accomplishing the syntactic mapping is usually tool-dependent rather than language-dependent, because the type of representations used depends on the capabilities of the modeling tool which will support the DSML (thus MOF provides no facility for specifying concrete syntax).

Each semantic domain and semantic mapping pair together specify a semantics for a DSML, and this semantics assigns a precise meaning to all of the well-formed models which obey the integrity constraints of the modeling language. Naturally, a single model might have multiple interesting interpretations; therefore a DSML might have a multitude of semantic domains and semantic mappings associated with it. For example, both structural and behavioral semantics are frequently associated with DSMLs. The structural semantics of a modeling language is a set-valued semantics which describes the meaning of the models in terms of the structure of model instances (all of the possible sets of components and their relationships, which are consistent with the well-formedness rules defined by the abstract syntax). Accordingly, the semantic domain for structural semantics is defined using sets. The behavioral semantics describes the evolution of the state of the modeled artifacts with respect to some abstraction of time. Hence, behavioral semantics is formally modeled by mathematical structures representing some form of dynamics.

In this paper, we will focus on metamodeling of the syntactic elements (A, C and $M_C : A \rightarrow C$) since they play the key role in tools and model transformations. Issues related to modeling semantics are discussed elsewhere[38].

Metamodeling Language Criteria

To effectively specify the syntax of DSMLs for graphical modeling tools such as GME, a metamodeling language should meet the following criteria:

- Provides sufficiently expressive yet generic object-oriented concepts capable of describing any conceivable domain.
- Enables specification of the diagrammatic representation of the domain concepts.
- Allows for the definition of the well-formedness rules for domain models.
- Includes some way to specify different logical views of domain models so modelers can focus on different relevant aspects of a system. This idea extends into the metamodeling language itself — the language should also include a similar facility for separating the concerns of the different interacting aspects of a DSML while it is being developed.
- Supports the extension, composition, and reuse of completed metamodels.

CHAPTER II

MOF OVERVIEW

The Meta Object Facility (MOF) is a sister-standard of UML and is maintained by the same standards-publishing body, the Object Management Group. MDA supplies both MOF and UML's profiling mechanism for defining specialized modeling languages; however, MOF is the true MDA metamodeling language - all the other MDA standards, including UML[16], CWM[19], and OCL[20], are specified using MOF[7]. A metamodel which is an instance of MOF formally specifies the abstract syntax of the set of modeling constructs which constitute a modeling language.

MOF is closely related to UML in that it utilizes the object-oriented UML Class Diagram constructs and uses them for modeling abstract syntax. For instances, MOF uses Classes to model domain concepts, Attributes to model concept properties, and Associations to model relationships between domain concepts. Consequently, MOF metamodels are similar to UML Class Diagrams. In fact, the UML Profile for Meta Object Facility defines a mapping between the elements of the MOF model and the elements of UML Class Diagrams, and it is possible to use this mapping to derive a graphical concrete syntax for MOF[23] (this mapping is useful because the MOF specification itself does not provide a concrete syntax for MOF[17]). MOF, however, strives to be simpler and smaller than UML Class Diagrams - the minimal metamodeling language.

The OMG ratified the earliest version of MOF in 1997. Prior to MOF, many previous attempts at model-based design had been centered on the assumption that a single object-oriented modeling language would be sufficient for modeling all types of data. These attempts failed because they did not take into account the fact that different types of computing systems require different types of modeling languages - no object oriented modeling language (or any other single type of modeling language) is universally applicable. MOF's fundamental premise, however, is that there must be multiple different types of modeling languages, each of which can provide a different view of computing systems[7].

MOF also has important relationships to two other MDA standards: XMI and CORBA. The XML Metadata Interchange specification, which was adopted one year after the initial adoption of MOF, provides a set of production rules which may be used to serialized any model defined by a MOF-specified metamodel into a standardized XML format[22]. XMI DTDs or schemas may be derived from metamodels; these are then used to validate the XMI documents generated from models. The OMG touts XMI as a standard format for interchanging all types of models between MOF-compliant modeling tools. There also exists a standard mapping between MOF and CORBA which defines the automatic generation of CORBA IDL-based interfaces from the abstract syntax specification of any modeling language defined using MOF. In addition to spelling out the syntax of these IDL interfaces, the MOF-CORBA mapping also enforces some of the API static semantics implied by the structure of a metamodel[17].

The MOF Architecture

MOF's architecture, or its overarching design and intended usage, conforms to the classic four-metalevel metamodeling framework[17]. Each metalevel in this framework consists of instances of elements of the next higher level.

- *M0 Level:* The concrete data of a system of interest at some point in time. Examples include the contents of a database or the execution of a finite state machine.
- *M1 Level:* The declarative model which defines a system using domain-specific concepts. One example would be a model of a specific finite state machine. Information at this level is also known as *metadata* because it describes the raw system data of level M0.

- *M2 Level:* The metamodel for the domain-specific modeling language capable of expressing the structure of a system's metadata. The UML model is a classic example; another example would be a language for modeling finite state machines.

- *M3 Level:* MOF, a self-describing meta-metamodel for specifying the abstract syntax of domain-specific modeling languages.

No metalevel beyond M3 is necessary to specify MOF, because MOF is self-describing (or metacircular)[17]. In essence, a metamodeling language such as MOF is simply a DSML for the domain of metamodels; as a specifier of all DSMLs, MOF is fully described using its own modeling concepts.

Basic MOF Concepts

As defined in the v1.4 specification [17], MOF provides the following five basic object-oriented concepts for use in specifying DSML abstract syntax:

- *Classes* are types whose instances have identity, state, and an interface. The state of a Class is expressed by its Attributes and Constants, and its interface is defined by Operations and Exceptions. Constraints can place limitations on the state of a Class.

- *Associations* describe binary relationships between Classes, including composition. Because MOF Associations have no object identity (that is, they are not first-class objects), they lack both state and interface. This deficiency makes the specification of some metamodels more awkward and difficult.

- *DataTypes* are non-instantiable types with no object identity. By design, the different MOF DataTypes encompass most of the CORBA IDL primitive and constructed types, including enumerations, structures, and collections.

- *Packages* are nestable containers for modularizing and partitioning metamodels into logical subunits. Generally, a non-nested Package contains all of the elements of a metamodel, so Packages are also the modeling concept responsible for enabling metamodel composition, extension, and reuse.

- *Constraints* specify the well-formedness rules which restrict the set of valid domain models.

Metamodel Composition and Reuse with MOF

MOF provides four features for metamodel composition, extension, and reuse: Class inheritance, Package inheritance, Class importation, and Package importation.

Both Classes and Packages can exist in OO-style generalization/specialization hierarchies which allow a derived Class (or Package) to inherit the structures and relationships of multiple base Classes (or Packages). Of course, Packages may not inherit from Classes and vice versa.

Package inheritance is MOF's facility for metamodel extension — a derived Package gains all of the metamodel elements defined in the Package from which it inherits. This facility is subject to constraints that disallow name collisions between inherited and locally-defined metamodel elements as well as name collisions between metamodel elements in the different base Packages in the case of multiple Package inheritance.

Class importation allows a Package to selectively acquire only the explicitly-desired types from another Package for use in Class inheritance, forming Associations, or defining new Attributes, Parameters, or Exceptions using the imported type.

Package importation is another feature for metamodel composition and reuse. It is semantically very similar to Package Inheritance, except that the modeling language described by the importing Package cannot be used to create instances of the Classes defined in the imported Package. However, the importing Package can subtype each of the Classes of the imported Package as if it had acquired them through Class importation, specify the types of typed elements such as Attributes using imported DataTypes, and define Operations which raise imported Exceptions.

MOF Technical Advantages

This section describes the technical advantages that MOF enjoys over MetaGME, the current GME native metamodeling language. MetaGME is discussed in more detail in Chapter 3.

Meaningful Class Operations

MOF provides the Operation, Parameter, and Exception concepts which may be used to model an interface to the operational semantics of a modeling language. This capability could be used to automatically generate full MIC model interpreter APIs. In the most current version of GME, the full interpreter API cannot be generated from a MetaGME metamodel - only methods which query the structure or state of domain models can be generated because MetaGME lacks the capability to model Class interfaces. Consequently, such interfaces must be added in by hand to the C++ code which can automatically be generated from a MetaGME metamodel[10].

Metamodel Composition and Reuse Facilities

GME provides a Library Import facility for the reuse of models (including metamodels) through extension. Library Import when applied to a metamodel closely resembles MOF's Package generalization feature. GME lacks any mechanism comparable to MOF's Package importation. However, it should be noted that while MOF's Package generalization disallows namespace conflicts between the base and derived Packages, such conflicts between and importing and an imported metamodel may be resolved in GME through the use of the Class Equivalence operator[3]. This operator enables the union of two metamodels along "join points" which are usually same-named metamodel elements.

Tool Independence

As a language, MOF is not dependent on any specific modeling tool technology. It does not mandate a concrete syntax. MOF is an industry standard which may be supported by many different modeling tools. Its metamodeling concepts are basic object-oriented concepts which may be intuitively grasped by any modern programmer. However, the Class stereotypes which form MetaGME's core modeling constructs are tightly bound to a set of core concepts which are used internally within GME. MetaGME has been structured with the clear assumption that the languages it specifies will be graphical in nature.

Before any tool can claim MOF compliance, it must have the capability to transform models to and from XMI as well as generate XMI DTDs or schemas for validating models serialized to XMI. Thus, MOF has the goal of enabling easy, reliable interchange of models between MOF-compliant tools. The fact that MOF effectively decouples the domain model from the tool with which it was built is another bonus of MOF's tool-independent nature.

MOF Technical Disadvantages

This section discusses disadvantages of using MOF to specify DSMLs.

MOF and DSML Concrete Syntax

As noted previously, MOF lacks any standard mechanism for specifying DSML concrete syntax. Thus, if a DSML requires a particular graphical notation, there is no standard way to declare that notation and map elements of the notation to elements defined in the metamodel which specifies the language's abstract syntax. Any tool-based solution which attempts to address this shortcoming by providing elegant support for concrete syntax specification in MOF would compromise the tool-independent nature of any metamodels thus specified - the concrete syntax information would likely be indecipherable to any other tool.

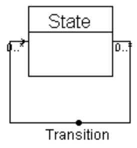

Figure II.1: Finite State Machine Metamodel

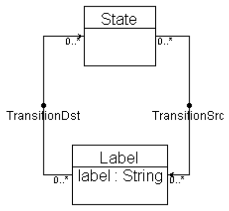

Figure II.2: Finite State Machine Metamodel with Labels

Lack of Association Classes

MOF's lack of support for Associations with state makes the definition of some DSMLs awkward. For example, consider the simple finite state machine metamodel above (Figure II.1). Finite state machines already have a well-defined graphical concrete syntax: States are represented as the circular nodes of a graph and Transitions between States are represented as directed arcs between the nodes. Each arc is labeled with a letter of the input alphabet to represent the event which prompts the traversal of the arc. But how do we model the fact that every Transition is associated with a label in MOF?

If MOF Associations had state, we could simply give the Transition Association a string-typed Attribute to store the letter from the input alphabet bound to each Transition instance. However, in MOF only Classes may have Attributes, so we must add a new Class, Label, to store this state. To model the fact that every Transition has a Label, we will need to divide our original Transition Association into two halves as shown in Figure II.2. Now, when a user wants to model a specific finite state machine, he or she will have to use twice the number of Associations as well as instantiating the Label Class for every Transition instance she wants to create between two State instances. Many DSMLs similarly require stateful Associations, and the inclusion of extra Classes to carry the burden of this state seems unnecessary and awkward.

Standard Immaturity

Due to a combination of the novelty of MOF and related standards and the lack of rigorous formality in their specifications, interoperability between “MOF-compliant” tools is not as dependable as hoped in the MDA vision. The MOF specification and the XMI specification both take the form of a series of diagrams supported by natural language descriptions. Due to the imprecise nature of natural language, each includes a number of ambiguities and contradictions which prevent the specifications from being implemented uniformly by modeling tool vendors. One result of this is that although the most current version of XMI is XMI 2.0, the de facto industry standard is *the Unisys implementation* of XMI 1.2.

CHAPTER III

GME OVERVIEW

Modeling Tool Architectures and Metaprogrammability

Modeling environments are typically supported by tool architectures which include several key tool components: a model builder, a model database, a constraint manager, and a number of model interpreter components[5]. Model builders expose some well-defined interface which allows modelers to perform standard CRUD (create, request, update, destroy) operations on model objects which are stored in the model database. The model builder must incorporate the concepts, relationships, composition principles, and representation formalisms of the supported modeling language, and the interface provided by the model builder may be textural, graphical, or mixed. The constraint manager is responsible for enforcing the well-formedness rules which restrict the set of valid models of a modeling language. Model interpreters 'execute' models in order to solve some problem of interest. When used, they translate system models into executable applications or input to analysis tools. System models, which serve as input to model interpreters, form the "problem space", a representation of that part of the world which is germane to some specific problem which must be understood and solved. The output of the model interpreters forms the "solution space", generally some executable simulation or analyzable mathematical formalism which provides useful information about the modeled system. In this sense, model interpreters may be said to implement the operational semantics of the supported modeling language. GME[2] is one example of a modeling tool which provides a model builder, model database, constraint manager, and model interpreter support; its architecture is depicted in Figure III.1.

MIC advocates the use of domain-specific modeling environments (DSMEs) because these environments are well suited for the design and implementation of complex CBSs. DSMEs such as MatLab[33] and LabView[30] have enjoyed great success, partially due to the large

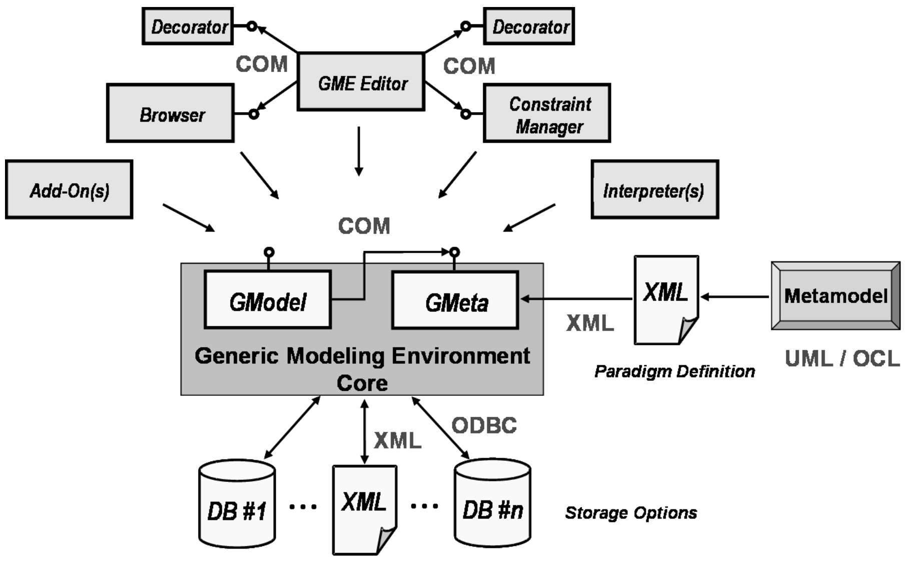

Figure III.1: Modeling Tool Architecture: GME

markets for the domains they capture. However, the primary drawback to the domain-specific approach is that building a new language and modeling tool to support a narrowly-used niche domain (for example, a co-design environment specialized for a single type of missile) might be unjustifiably expensive. At the other extreme, a general modeling tool with "universal" modeling concepts and components would lack the primary advantage offered by the MIC approach: dedicated, customized support for a wide variety of application domains. A third alternative is to use a highly-configurable modeling tool with model building, constraint management, and model interpretation components which may be easily customized to support a unique environment for any given application domain.

Metaprogrammable tools such as the Generic Modeling Environment (GME) rely on metaprogramming as the mechanism for accomplishing this level of configuration[4]. For example, GME incorporates a generic set of graphical model-building idioms, a constraint manager, and a database backend. Users may draw upon the graphical idioms to implement the structure and representation of domain objects and relationships. They may also configure the constraint manager to enforce the particular well-formedness rules of the domain. Finally, they may configure the database backend for storing domain model objects. Metaprogramming is the process through which the user configures the (A) metaprogrammable model builder by mapping a DSML into GME's graphical idioms, (B) the constraint manager by assigning constraints to the valid domain modeling operations, and (C) the database backend through the specification of an appropriate database schema. The result of this metaprogramming is a valid DSME supported in GME[27]. Other metaprogrammable tools include Dome[24] and MetaEdit+[6].

Origins of Metaprogramming: The Multigraph Architecture

The evolution of GME begins in 1995 with the Multigraph Architecture (MGA), one of the earliest metaprogrammable modeling tools[27][28]. MGA concentrated on supporting model-based design environments for large-scale embedded computing applications dominated by mature engineering disciplines. Essentially, the operation of MGA divided into three layers: the meta-level layer for the specification of DSMEs, the domain-specific modeling layer, and the model execution layer. Early versions supported the use of an early informal declarative metaprogramming language at the meta-layer which introduced several of the fundamental modeling patterns which dominate GME's current native metamodeling language. These patterns included aspects (model views), associations, membership-based groupings, hierarchically-composable entities (modules) with connection interfaces, integrity constraints, module interconnections, and specialization. MGA conceptualized DSMEs as unique combinations of these model composition principles. This metaprogramming language effectively supported the informal specification of the abstract syntax, concrete syntax, syntactic mapping, and static semantics (domain constraints) of DSMLs. The MGA meta-level also included a set of meta-level translators to automatically generate configuration files for its model builder and model database directly from DSME specifications. Note, however, that the MGA's earliest metaprogramming language was not actually a metamodeling language — it was simply an informal declarative configuration language used to generate configuration files. Because the language was not formal, there was no way to validate the consistency of the specified modeling concepts to ensure that the specified environment sufficiently constrained domain modelers[13]. Thus, MGA's structure incorporated only the bottom three of the four meta-levels advocated in OMG's modeling formalism.

Metamodeling Extensions to MGA

In 1998, support for the design of DSMEs through metamodeling was added to MGA in order to address the shortcomings of the earlier version's metaprogramming language[14]. The adopted formal metamodeling language consisted of UML Class Diagrams with OCL constraints. Thus, the abstract syntax of MGA DSMEs could be formalized as UML-based metamodels through graphical entity-relationship diagrams, and domain constraints could be formalized in OCL as textual invariant Boolean expressions. This MGA metamodeling approach required the specification of DSML concrete syntax as a separate step before a metamodel could be used to synthesize an MGA modeling environment. The specification was accomplished by mapping the entities and relationships specified in the class diagram to various MGA-specific presentation objects and patterns. These objects and patterns included general model composition abstractions, such as hierarchy and aspects, which were carried over from the previous version of MGA.

This transition represented not only an advance in the formality of the MIC model-based design process, but also a leap forward in the usability of MGA. By using industry-standard modeling languages for metaprogramming, potential MIC users familiar with those modeling languages could more quickly and accurately communicate their DSME requirements to MGA metamodelers. The new approach separated the concerns of abstract syntax, concrete syntax, and static semantics which had been tangled in the previous metaprogramming language. Metamodeling also opened the door both to metamodel reuse and to inter-tool transfer of modeling language specifications. Unfortunately, the mapping of the class diagram into an MGA-specific concrete syntax specification proved to be a bottleneck — the UML class diagrams did not contribute much to the actual definition of modeling environments, and concepts such as inheritance had to be enforced by hand. Consequently, the DSME specification process was still error-prone and slow.

Modern Metamodeling with GME

1999 marked a substantial revision of both the MGA core modeling constructs and the MGA metaprogramming facilities in an effort to make MIC solutions easier to implement[25]. The core MGA constructs underwent two types of changes: First, several constructs changed in name (for instance, the membership-based grouping construct's name was changed from "Conditional Controller" to "Set"). Second, several constructs which had lacked object identity in the previous version of MGA became first-class objects, including References, Connections, and Sets. Thus, these constructs became configurable in name and representation.

The metaprogramming facilities had to evolve to support these changes, but also to increase usability and expressiveness. A boot-strapping process was used to develop a CBS to serve as the metamodeling environment for the new MGA tool, the Generic Modeling Environment (GME). This new metamodeling environment, called MetaGME, still consists of UML Class Diagrams and OCL constraints, but unifies the functionality of the two diagrams needed to specify abstract and concrete syntax in the previous MGA implementation. MetaGME, like MOF, is meta-circular — it has been used to model itself and is self-defined using its own concepts. The model of GME's metamodeling environment is GME's meta-metamodel. This approach allows some further improvement by evolving the meta-metamodel and then boot-strapping the improvements into the metamodeling environment. However, this is still not a trivial task even with the presence of the reconfigurable meta-metamodel because GME must maintain the ability to support all existing metamodels. This factor strongly influenced the approach taken in building a MOF metamodeling environment for GME.

GME is currently the flagship tool of MIC. In addition to serving as a metaprogrammable modeling tool in the spirit of the original MGA, GME provides library import and export facilities and custom model visualizations. There are also facilities for plugging in analysis, verification, and translation tools which interpret domain-specific models. GME's metamodeling language, MetaGME, utilizes UML stereotypes to imply part of the syntax expressed by the metamodel. It also maintains the important modeling patterns which have been central to MGA since its earliest versions, such as module interconnection. The meanings of the stereotypes used in MetaGME are[4]:

- *Models* are hierarchically-compound modular objects.
- *Atoms* are elementary, non-decomposable objects.
- *FCOs* are generic first-class objects which must be abstract but can serve as the base type of an element of any other stereotype in a specialization relationship.
- *References* refer to other model objects.
- *Connections* are analogous to UML Association Classes.
- *Aspects* provide logical visibility partitioning to present different views of a model.

GME-based metamodeling is demonstrated in Figure III.2. The metamodel ${}_{MetaGME}MM_{DSML}$ of a DSML consists of the abstract syntax ${}_{MetaGME}A_{DSML}$, concrete syntax ${}_{MetaGME}C_{DSML}$, and syntactic mapping ${}_{MetaGME}M_{CDSML}$ specified using the UML constructs of MetaGME. The ${}_{MetaGME}MM_{DSML}$ metamodel is translated by the T_1 meta-level translator (called the meta-interpreter) into a configuration file for GME (represented in Figure III.2 by the box labeled "GME/Meta"). Using this configuration file, GME configures

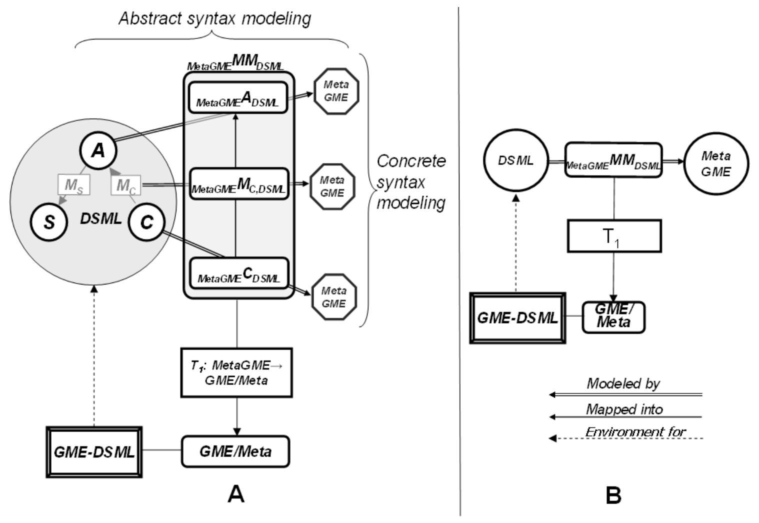

Figure III.2: Metamodeling with GME

its graphical model builder, model database, and constraint manager to function as the domain-specific modeling environment for the metamodeled domain. (A simplified diagram of the metamodeling process can be seen in Figure 1.B)

There exists a large body of existing GME-based DSML-s[34][8]. There also exist a number of related modeling tools, including the model-to-model transformation tool GReAT[1]. Because of this large volume of existing DSMLs and tools which depend on the existing GME metamodeling language, a whole-sale replacement of MetaGME is not desirable. This thesis describes work to update the MIC metamodeling facilities to incorporate the MOF standard alongside the UML/OCL-based MetaGME.

CHAPTER IV

GME-MOF

This chapter describes the implementation of a functional MOF-based metamodeling environment for metaprogramming GME. GME-MOF leverages the existing GME metamodeling language and meta-level translators for the realization of new GME DSMEs[32][31].

Solution Overview

As described previously, any functional GME metaprogramming system must incorporate two key components. The first component is a metamodeling environment (ideally, a graphical environment) which supports the specification of abstract syntax, concrete syntax, and syntactic mappings. The second component is a translation tool capable of generating from the metamodel of a target domain the configuration file which customizes GME to serve as the DSME for that domain. MetaGME itself defines a metamodeling environment with sufficient expressive power to fully model MOF. So, while the MOF specification uses MOF to model itself, GME-MOF includes a new GME metamodeling environment which expresses the MOF model using MetaGME's constructs. Furthermore, because MetaGME by design already reflects the full range of configurations which can be realized by GME, the easiest way to acquire the necessary translation tool is by defining a model-to-model transformation algorithm from MOF-specified metamodels into MetaGME-specified metamodels.

Transforming a MOF-specified metamodel into a MetaGME metamodel enables the conscription of MetaGME's existing meta-interpreter to generate the GME configuration file. The transformation algorithm is quite straightforward because both MOF and MetaGME are sister-languages of UML Class Diagrams. GReAT was the natural choice for implementing this metamodel translation algorithm - it enabled the easy and rapid creation of the executable metamodel translation tool, named MOF2MetaGME. MOF2MetaGME itself is easy to analyze, maintain, and evolve as the two languages it bridges evolve.

The shaded components in Figure IV.1 represent the new facilities required to implement MOF for GME: ${}_{MetaGME}MM_{MOF}$ is the MetaGME-specified MOF model and T_2 is the MOF2MetaGME transformation algorithm from MOF-specified metamodels to MetaGME-specified metamodels. T_1 is MetaGME's meta-interpreter, the meta-level translator which generates the GME configuration files from the translated metamodels. This configuration file customizes GME's graphical model builder, constraint manager, and model database to support the modeled DSME.

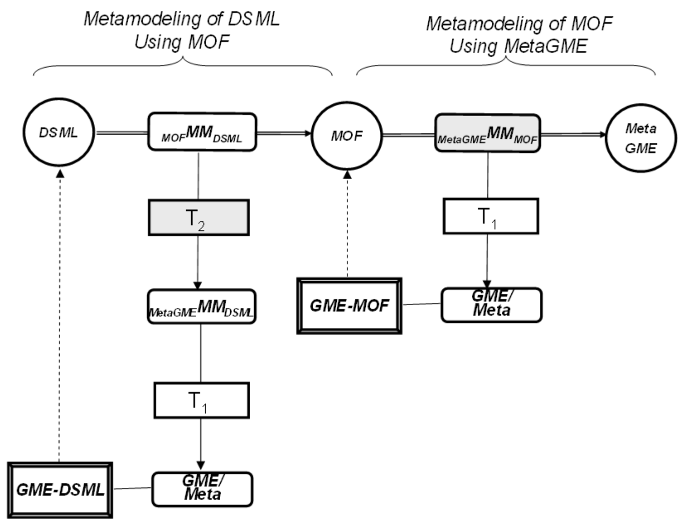

Figure IV.1: Building the MOF-Based Metamodeling Environment

Appendix A describes the implementation of the GME-MOF metamodeling environment, and Appendix B describes the implementation of MOF2MetaGME.

Figures IV.2, IV.3, and IV.4 compose a small example which illustrates the full function of GME-MOF. Figure IV.2 is a small part of a MOF-based implementation of UML class diagrams used as the input to MOF2MetaGME, and Figure IV.3 is the corresponding output produced by MOF2MetaGME. Note the high degree of symmetry between the two diagrams. Figure IV.4 demonstrates the graphical DSME interpreted from the MetaGME metamodel shown in Figure IV.3.

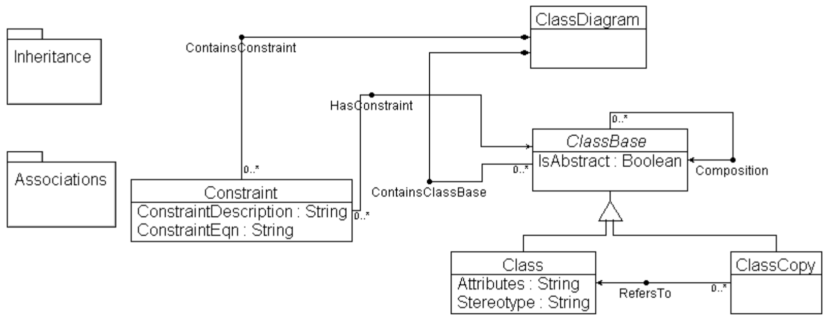

Figure IV.2: UML Class Diagrams in MOF

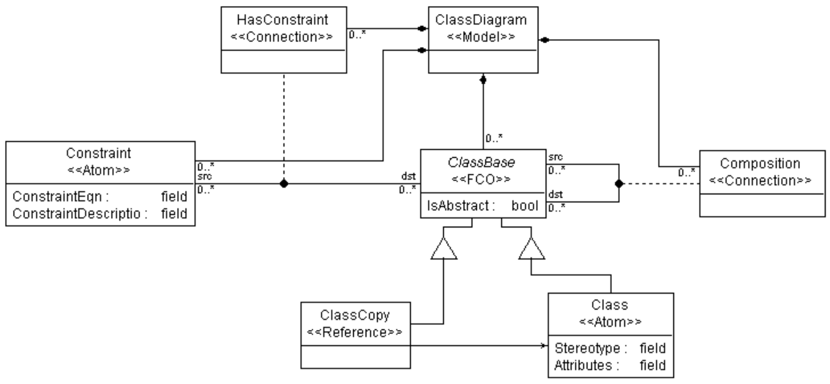

Figure IV.3: UML Class Diagrams in MetaGME

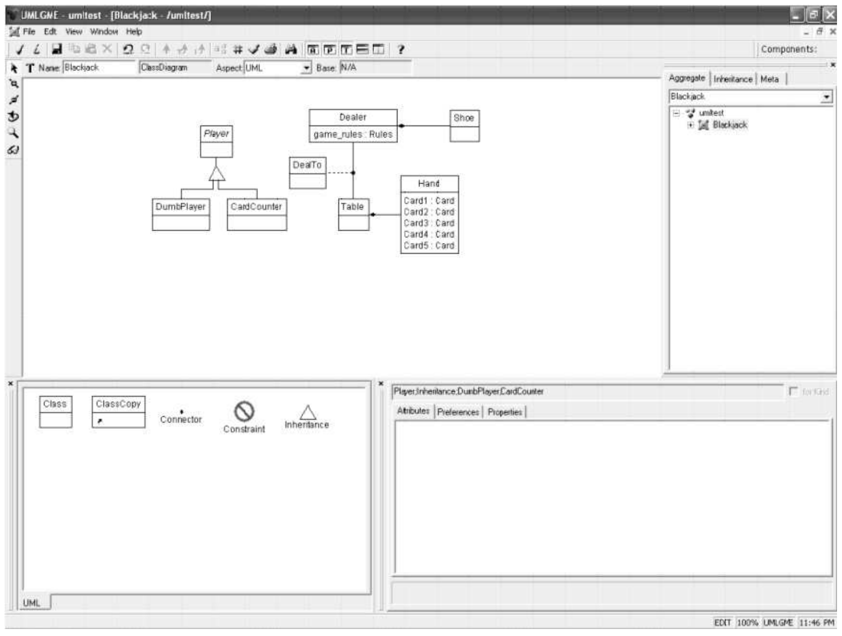

Figure IV.4: UML Class Diagrams DSME

Solution Advantages

GME-MOF successfully enables MOF-style metamodel specification in GME by reusing the entire existing GME metaprogramming toolset. Consequently, it does not require any major overhaul of the GME core constructs which would break compatibility with existing GME DSMEs. GME-MOF implements MOF as an additional layer of abstraction over MetaGME.

Furthermore, GME-MOF takes advantage of MOF's light-weight extension mechanism, the Tag, to include some GME-specific syntax information in MOF metamodels. The GME-MOF metamodeling language includes augmented MOF Classes, Associations, Packages, Constraints, and Attributes with additional fields which may be conceptualized as MOF Tags. These fields specify GME tool-specific information and facilitate mapping into MetaGME. In this way, GME-MOF allows the specification of concrete syntax without deviating from the MOF standard.

GME also supports an extension mechanism similar to MOF Tags through the GME Model Registry. Like MOF Tags, GME registry entries are simple name-value pairs. MOF2-MetaGME makes use of the GME registry to store encoded information about the MOF DataTypes, Operations, Parameters, and Exceptions expressed in a MOF metamodel when it performs the transformation into MetaGME. Then, the BONExtender, a meta-level translator which generates domain-specific model interpreter C++ APIs from MetaGME metamodels, can recover this information and include it in the generated API. Usually these operations, exceptions, and data types would need to be hand-woven into the automatically-generated C++ class definitions rendered by the BONExtender, which is a tedious and brittle process. In this way, users tangibly benefit from modeling Class operations and data types in their MOF metamodels, even though operation modeling is not supported by MetaGME. This feature allows more model interpreter code to be autogenerated from models rather than being hand-written.

Solution Limitations

The limitations of GME-MOF largely stem from the dissonance between MOF as a tool-independent metamodeling language and MetaGME as a metamodeling language tightly coupled to a graphical modeling tool. The translation from MOF into MetaGME is not isomorphic — MOF provides some constructs and capabilities that MetaGME lacks (and vice-versa). MOF allows a wider range of potential attribute types, the concepts of derived attributes and associations, singleton classes, and classifier-scoped attributes. None of these concepts are supported for domain modeling in GME. Likewise, MetaGME provides facilities for multi-view modeling, concrete syntax specification, and some syntax identifiers which carry special meaning as GME graphical modeling idioms. As a result of these differences, users can construct valid MOF metamodels which cannot be fully rendered as GME DSMEs. In these cases, the MOF2MetaGME translator simply discregards the use of any feature which cannot be mapped into features supported by MetaGME.

Additionally, neither MOF nor MetaGME are stable languages. The OMG will soon release an updated version of MOF, MOF 2.0[18]. MetaGME itself constantly evolves in small ways in response to user requests and to maintain synchrony with the internal GME modeling constructs. Consequently, both the GME-MOF metamodeling environment and the MOF2MetaGME transformation require consistent updates to support the most current versions of both languages. Fortunately, both GME metamodels and GReAT transformations were designed with system evolution in mind.

Case Study and Evaluation

This section describes the specification of a simple DSME for hierarchical finite state machines (HFSM) using GME-MOF. The metamodel for this DSME is evaluated in comparison to a HFSM metamodel built natively using MetaGME.

HFSM in GME-MOF

The GME-MOF metamodel for HFSM is built from three packages (Figure IV.5):

- *Primitives,* which merely stores primtive DataType definitions such as String and Integer
- *Event,* which contains a simple metamodel for fully-ordered sequences of events
- *HFSM,* which defines hierarchical finite state machines by reusing the definitions from the *Event* package.

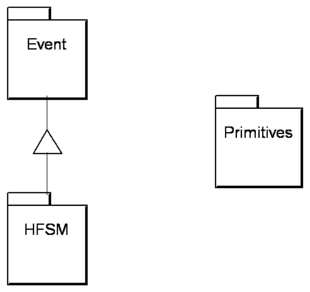

Figure IV.5: Packages used to define HFSM

The internals of the *Event* package appear in Figure IV.6. This metamodel specifies that event instance may be created with instances of InputSequence and may form Sequence relationships with one another. The Sequence relationship may be used to provide a total ordering of the Events in an EventSequence, as the multiplicity settings of Sequence specify that each Event may be directly preceded by at most one other Event. The metamodel uses the Delay attribute of Event to capture any time delay which should occur between Events. Delay is typed using the Integer PrimitiveType imported from the *Primitives* package. Tags were used to tailor this metamodel for transformation into MetaGME to achieve the following effects:

- Event maps into a MetaGME Atom.

- InputSequence maps into a MetaGME Model.
- The Sequence association maps into a MetaGME Connection contained by the Input-Sequence Model.
- The InputSequence Model may appear in the Root Folder of a project.
- Event uses the icon 'event.bmp' for its concrete syntax.

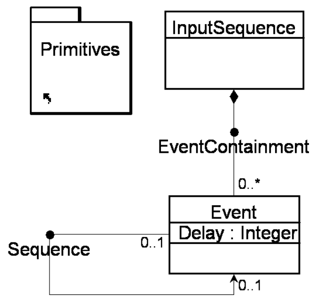

Figure IV.6: The *Event* Package

The metamodel encapsulated by the *HFSM* package appears in Figure IV.7. It describes hierarchical States connected by Transitions, including special final and initial states. *HFSM* inherits from *Event* to allow users to model the events which drive a given state machine. There are several interesting things to note regarding this metamodel. First, because Transition requires some state, it must be declared as a class and not as a simple association. Second, the Event attribute of Transition is the imported String type instead of the Event type inherited from the *Event* package. This is a concession to GME, which only supports attributes of type integer, double, float, string, and boolean. In order to build a correspondence between the event used by a Transition instance and the Event instances declared in a model, an OCL constraint is defined stating that the value of a Transition instance's event attribute must be the same as the name of some Event instantiated in the model. Finally, State defines an operation, is_reachable, which takes in two States, start and dest, and returns a Boolean. MOF does not allow modeling of the operational semantics of this operation, but a model interpreter might define it to perform reachability analysis on a hierarchically-embedded state machine. Tags were used to tailor this metamodel for transformation into MetaGME to achieve the following effects:

- State maps into a MetaGME Model.
- The State Model may appear in the Root Folder of a project.
- TransitionOut and TransitionIn map into MetaGME Connections contained by the State Model.
- Transitions map into MetaGME Atoms.
- State, FinalState, InitialState, and Transition use icons 'state.bmp', 'final.bmp', 'initial.bmp', and 'event.bmp' respectively for concrete syntax.

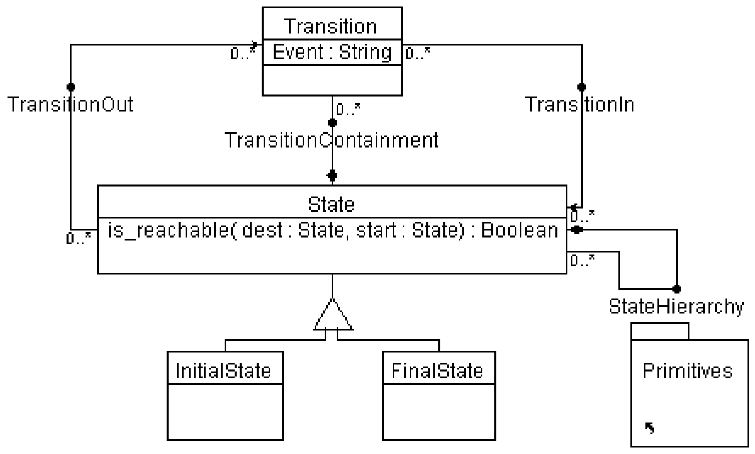

Figure IV.7: The *HFSM* Package

Evaluating GME-MOF versus MetaGME

A similar MetaGME metamodel for simple HFSM consists of two the ParadigmSheets displayed in Figures IV.8 and IV.9.

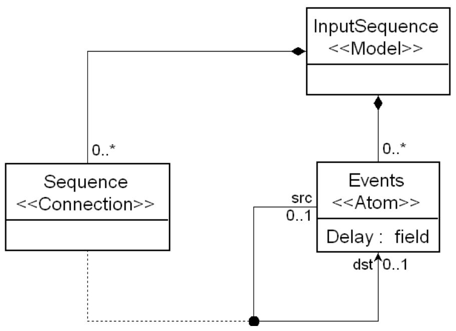

Figure IV.8: The *Event* ParadigmSheet

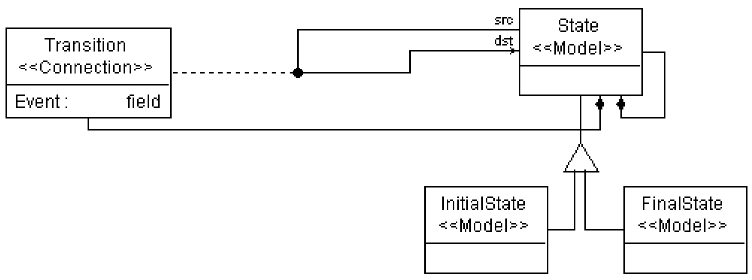

Figure IV.9: The *HFSM* ParadigmSheet

Comparing this metamodel to the metamodel used to build HFSM in GME-MOF can provide a practical evaluation of the relative strengths and weaknesses of GME-MOF. The HFSM case-study illuminates three primary differences in the metamodeling capabilities of the two environments:

1. The GME-MOF version of HFSM includes the explicit modeling of a class operation, is_reachable, using first-class language constructs. This eases the specification of a domain-specific API which could be used by a model-interpreter to provide analysis capabilities for HFSM models. MetaGME lacks language constructs which deal with operations.

2. Because MetaGME is the native metamodeling language of GME, there is no need to keep any lower-level metamodeling language in mind while metamodeling. For

example, in MetaGME, it is natural to model the Event attribute of Transition as a string-typed attribute, because that is all that MetaGME allows. In order to hold consistency with the MOF standard, GME-MOF allows another option: the Event attribute of Transition could have type Event. However, the metamodeler must keep the restrictions of MetaGME in mind while metamodeling in GME-MOF.

3. As discussed previously in this thesis, the lack of association classes in MOF makes the specification of stateful relationships somewhat awkward. This awkwardness manifests itself in the definition of Transition in GME-MOF as a class joined to State by two associations versus the definition of Transition in MetaGME as a single stateful connection.

Although GME-MOF is not entirely an improvement on MetaGME, the HFSM example illustrates that it provides a useful technical advantage (class operation modeling) while allowing useful DSMEs to be specified for GME using a MOF-compliant language.

CHAPTER V

DISCUSSION AND FUTURE WORK

This thesis describes work to implement an alternative MOF-based metamodeling environment for a metaprogrammable modeling tool through metamodeling and model-to-model transformation. The solution depends on a model transformation from MOF-specified metamodels to analogous metamodels in a different, tool-specific metamodeling language. The approach is useful because it enables the reuse of the tool's existing metaprogramming facilities and avoids breaking its compatibility with existing DSME and models. However, there are drawbacks to this solution as well. One is that the transformation between the two metamodeling languages must be constantly updated, as both languages will continue to evolve in response to user demands. Another is that the two metamodeling languages have enough differences that neither can be elegantly mapped into the other. These concerns give rise to two important questions:

- Do complicated metamodeling languages like MOF or MetaGME make good "core" metaprogramming languages for metaprogrammable modeling tools?
- How can we design adaptable metaprogrammable tool architectures to minimize the effect of changing metamodeling languages?

Instead of basing metaprogrammable tools on a high-level, user-friendly metamodeling language, we might use a minimal language expressing a core abstract metaprogramming semantics meaningful to any metaprogrammable modeling tool. Alternatively, we might use a pair languages — one for describing abstract syntax and another for describing concrete syntax. Because the general requirements of metaprogrammable modeling tools (entity-relationship modeling, support of CRUD operations on models, constraint management, etc...) do not change, this minimal metaprogramming language could be stable and standardized. Then, we could express the semantics of more complicated, user-friendly languages such as MetaGME and MOF in the primitive constructs defined by the minimal metaprogramming language. This approach could yield two primary benefits: **1)** The minimal metaprogramming language would define the limits of what features a higher-level metamodeling language might provide. If some feature cannot be rendered down and expressed by the abstract metaprogramming semantics, then it should not be in a metamodeling language. **2)** It would easier to define a model-interchange standard for the simple minimal metaprogramming language than for a complicated language with many constructs such as MOF. One such commercial metamodeling facility which attempts to use this approach is XMF[36][37]. This language provides a simplified MOF-like language for modeling abstract syntax. This core language is then extended with a seperate concrete syntax modeling language, OCL for specifying well-formedness rules, an action language called XOCL for modeling operational semantics, and a mapping language called XMap for specifying model-to-model transformations. All of XMF is ultimately defined using an even simpler core set of executable metamodeling constructs called XCore.

We might also consider generalizing the solution presented in this thesis — model-to-model transformations can mitigate some of the complexity of designing and interfacing the various components of an adaptable metaprogrammable tool architecture. Solutions implemented through graphical model-to-model transformations are easier to design, build, understand, and evolve than functionally-similar traditional programming solutions. Model-to-model transformation languages such as GReAT can leverage the power of domain-specific modeling to decouple the metamodeling language from the other components of a metaprogrammable modeling tool. For example, consider the model interchanger, a common modeling tool component which converts models both to and from some model interchange language. Model interchange languages allow the migration of models between tools. MOF, for instance, provides a mapping to XMI, the OMG standard model interchange language. This component can be most easily implemented as a model-to-model translation between meta-models or domain models and models in the interchange language. Such a transformation could be easily maintained as the relevant standards change and evolve.

APPENDIX A

GME-MOF ENVIRONMENT IMPLEMENTATION

This appendix provides an abbreviated specification of the GME-MOF environment (minus the details of the UML Class Diagrams-like concrete syntax and the actual OCL constraint equations) in the form of a series of MetaGME class diagrams, natural language constraint descriptions, EnumAttribute enumeration labels, and Aspect visualization information. Detailed information about MetaGME may be found in the GME User's Manual[10].

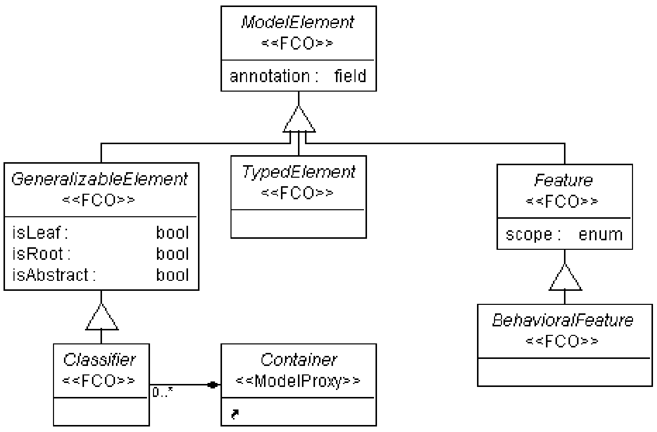

Figure A.1: Abstract Base Classes

Abstract Base Classes (Figure A.1)

Constraints:

Name: MustHaveType

Constrains: TypedElement

Description: A TypedElement must have one and only one type.

Visualization:

TypedElement and BehavioralFeature are visible in the Features Aspect.

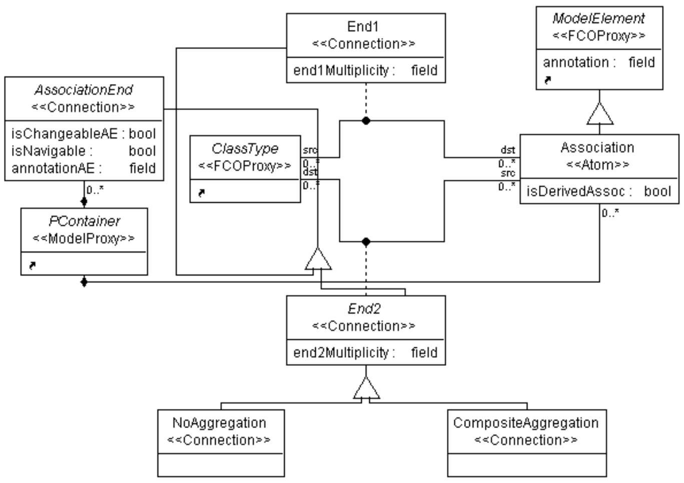

Figure A.2: Association

Association (Figure A.2)

Constraints:

Name: BinaryAssociations

Constrains: Association

Description: Associations must be binary.

Name: NoNameCollisions

Constrains: Association

Description: The contents of a Namespace may not collide.

Visualization:

Associations are visible in the ClassDiagram Aspect.

Class, Attribute, and Operation (Figure A.3)

Constraints:

Name: NotSingletonAndAbstract

Constrains: Class

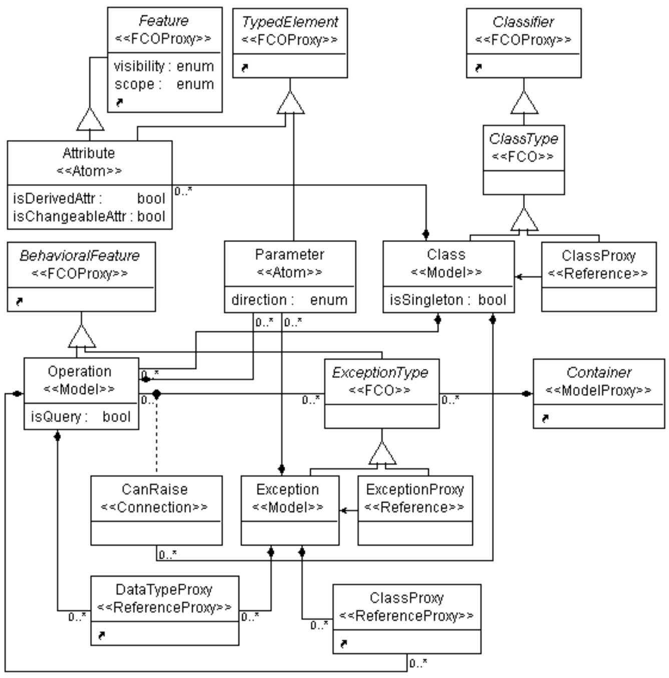

Figure A.3: Class, Attribute, and Operation

Description: A class may not be both singleton and abstract.

Name: AllOutParam

Constrains: ExceptionType

Description: An Exception's Parameters must all have the direction 'out'.

Name: OneReturnParam

Constrains: Operation

Description: An Operation may have at most one Parameter whose direction is 'return'.

Name: NotNull

Constrains: ClassProxy, ExceptionProxy

Description: A proxy may not be null.

Name: LegalProxy

Constrains: ClassProxy, ExceptionProxy

Description: This element must be visible in the current context before it can be proxied.

Enumeration Labels:

Parameter::direction: in, out, inout, return

Visualization:

ClassType is visible in the ClassDiagram Aspect. ClassType and CanRaise are visible in the Features Aspect.

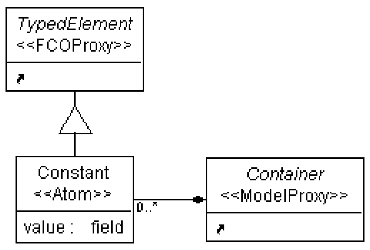

Figure A.4: Constant

Constant (Figure A.4)

Constraints:

Name: TypeIsPrimitive

Constrains: Constant

Description: Constants must have primitive types.

Constraint (Figure A.5)

Constraints:

Name: ValidElement

Constrains: Constraint

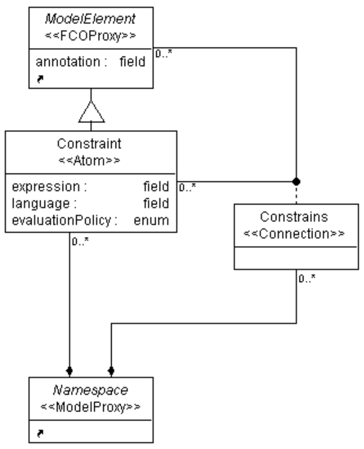

Figure A.5: Constraint

Description: Constraints, Imports, Tags, and Constrants may not be constrained.

Enumeration Labels:

Constraint::EvaluationPolicy: immediate, deferred

Visualization:

ModelElement, Constraint, and Constrains are visible in the Constraints Aspect.

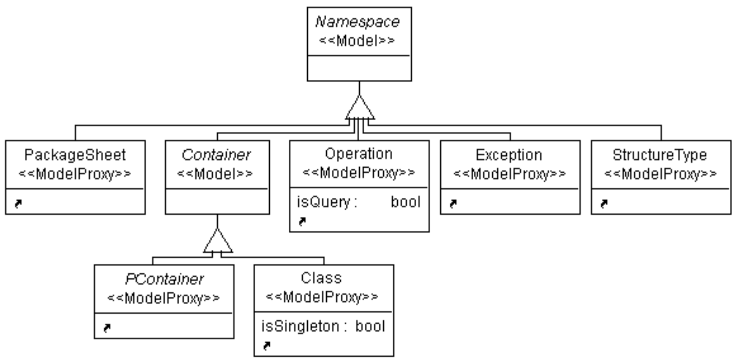

Figure A.6: Containment

Containment (Figure A.6)

Constraints:

Name: NoNameCollisions

Constrains: Namespace

Description: The contents of a Namespace may not collide.

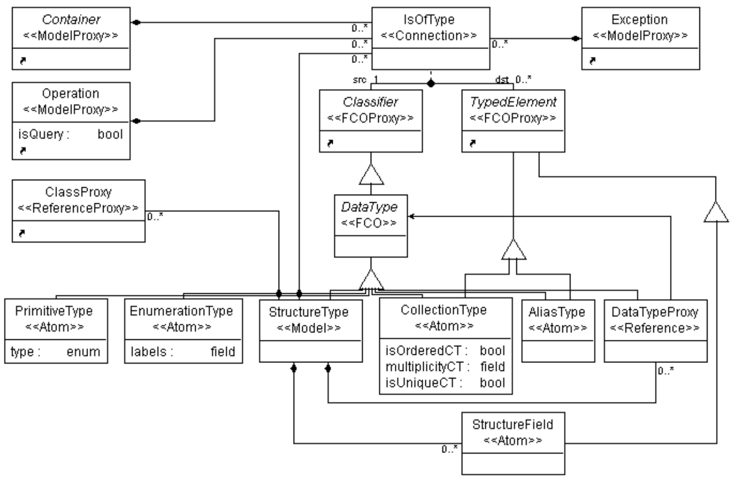

Figure A.7: DataType

DataType (Figure A.7)

Constraints:

Name: NotAbstract

Constrains: DataType

Description: A DataType cannot be abstract.

Name: ContainsStructureField

Constrains: StructureType

Description: A StructureType must contain at least one StructureField.

Name: NotNull

Constrains: DataTypeProxy

Description: A proxy may not be null.

Name: LegalProxy

Constrains: DataTypeProxy

Description: This element must be visible in the current context before it can be proxied.

Name: NotProxied

Constrains: DataTypeProxy

Description: A DataType Proxy must reference a type, not another proxy.

Visualization:

DataType and IsOfType are visible in the Features Aspect.

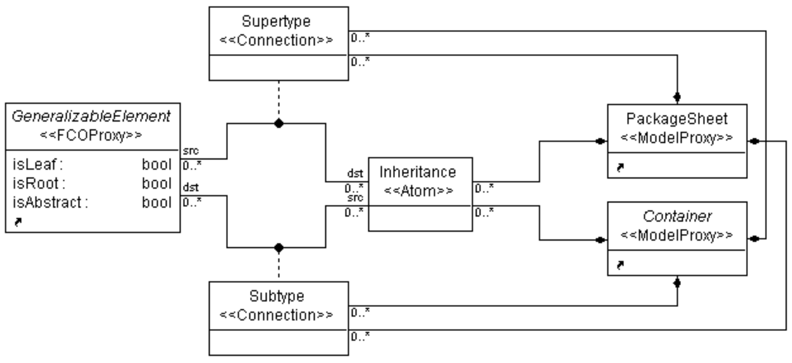

Figure A.8: Generalization

Generalization (Figure A.8)

Constraints:

Name: HasDerived

Constrains: Inheritance

Description: Inheritance operator is superfluous or invalid. It must have a derived element.

Name: AllowableType

Constrains: Inheritance

Description: Only Classes and Packages may participate in generalization relationships.

Name: SingleBase

Constrains: Inheritance

Description: Inheritance operator is superfluous or invalid. It must have one and only one base element.

Name: NoRecursion

Constrains: GeneralizableElement

Description: Recursive inheritance chains are not allowed.

Name: RootCannotGeneralize

Constrains: GeneralizableElement

Description: Root elements cannot be generalized.

Name: LeafCannotSpecialize

Constrains: GeneralizableElement

Description: Leaf elements cannot be specialized.

Name: NoAncestorNameConflicts

Constrains: GeneralizableElement

Description: The names of the contents of the supertypes of a GeneralizableElement may not collide with one another.

Name: NoInheritedNameConflicts

Constrains: GeneralizableElement

Description: The names of the contents of a GeneralizableElement should not collide with the names of the contents of any direct or indirect supertype.

Name: NoIllegalDependencies

Constrains: GeneralizableElement

Description: The base type of a GeneralizableElement must lie within the scope of the derived type.

Visualization:

Supertype, Subtype, and Inheritance are visible in the ClassDiagram Aspect.

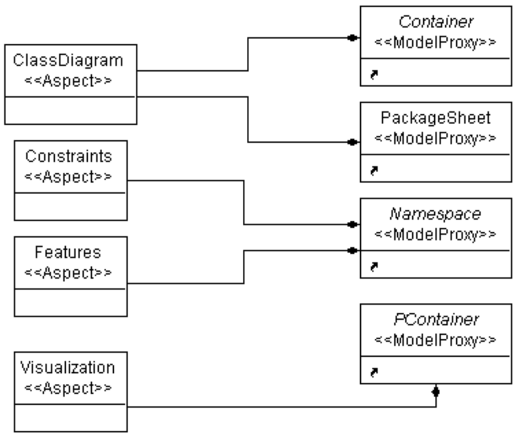

Figure A.9: GME-MOF Aspects

GME-MOF Aspects (Figure A.9)

No constraints, labels, or visualization information.

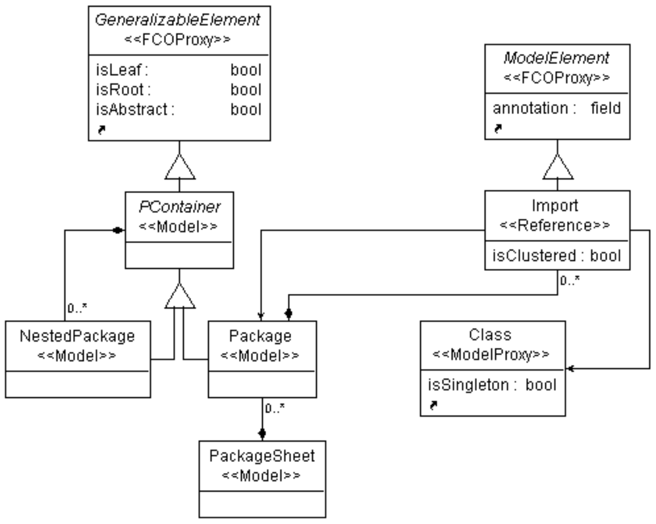

Figure A.10: Package

Package (Figure A.10)

Constraints:

Name: NotAbstract

Constrains: PContainer

Description: A package may not be declared abstract.

Name: NotEmpty

Constrains: PContainer

Description: Package is invalid or superfluous. It contains nothing.

Name: CannotImportSelf

Constrains: Import

Description: Packages cannot import or cluster themselves.

Name: SingleSheet

Constrains: PackageSheet

Description: There can only be one PackageSheet in a project.

Name: NotNull

Constrains: Import

Description: An Import may not be null.

Name: CannotImportContents

Constrains: Import

Description: Packages cannot import or cluster Packages or Classes that they contain.

Visualization:

Import, PContainer, and PackageSheet are visible in the ClassDiagram Aspect.

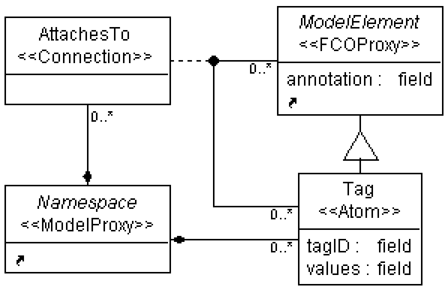

Figure A.11: Tag

Tag (Figure A.11)

Visualization:

Tag and AttachesTo are visible in both the ClassDiagram and Features Aspect.

Multi-Aspect Modeling (Figure A.12)

Constraints:

Name: ModelsHaveAspects

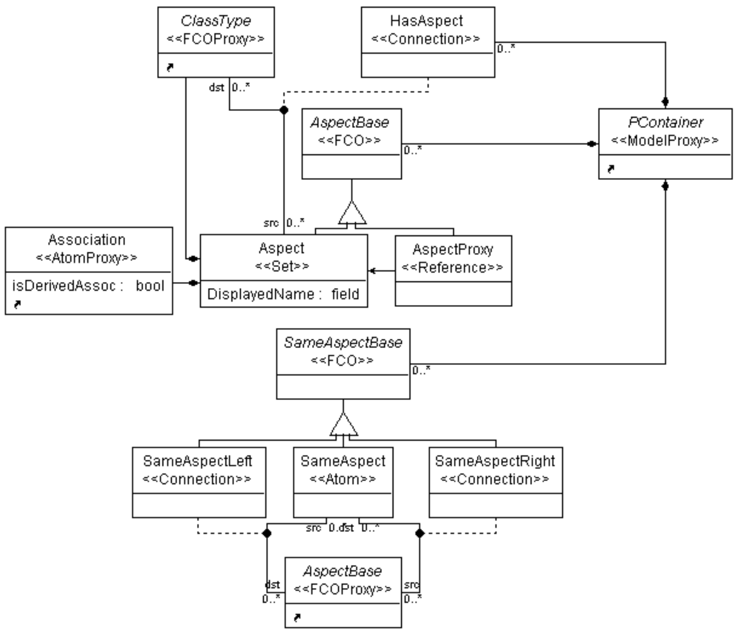

Figure A.12: Multi-Aspect Modeling

Constrains: ClassType

Description: Only Classes of GME Stereotype “Model” may have Aspects.

Name: MustHaveOpenAspect

Constrains: ClassType

Description: Classes of GME Stereotype “Model” must have at least one open Aspect.

Name: HasMember

Constrains: Aspect

Description: An Aspect must have at least one Class member.

Name: NotNull

Constrains: AspectProxy

Description: A proxy may not be null.

Name: OneRight

Constrains: SameAspect

Description: The SameAspect operator must have one and only one right operand.

Name: OneLeft

Constrains: SameAspect

Description: The SameAspect operator must have one and only one left operand.

Name: ValidOperands

Constrains: SameAspect

Description: One of the operands of the SameAspect operator must be an AspectProxy

Visualization:

SameAspectBase, Association, ClassType, AspectBase, and has HasAspect are visible in the Visualization Aspect.

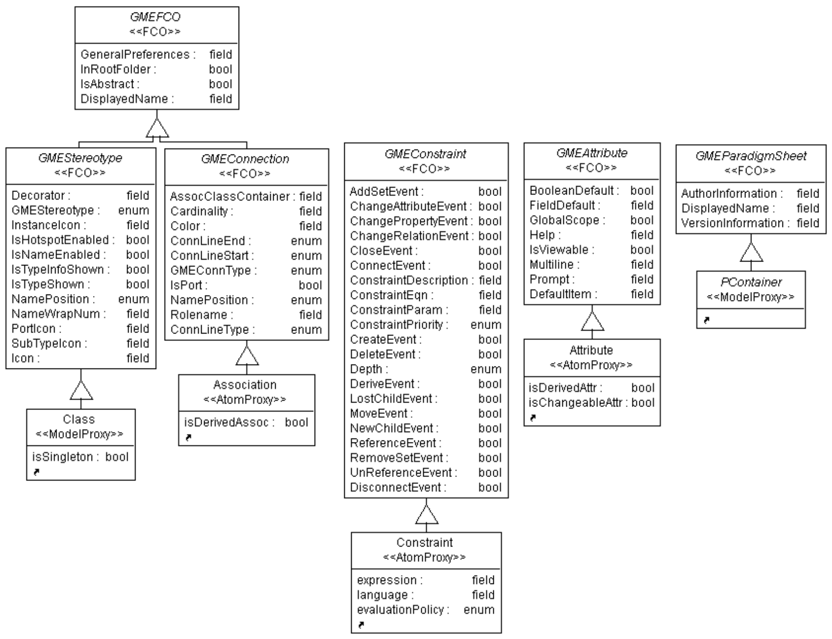

Figure A.13: GME Mappings

GME Mappings (Figure A.13)

Note regarding GME Mappings: No constraints, labels, or visualization information apply to these constructs. Inheritance is used to augment some MOF elements with the ability to specify information relevant to GME, including concrete syntax specifications. The definitions of each of these additional attributes are given in the GME Manual and User Guide[10]. Note that these attributes may be conceptualized as MOF Tags applied on an element-by-element, metamodel-by-metamodel basis.

APPENDIX B

THE MOF2METAGME TRANSFORMATION

Essentially, MOF2MetaGME expresses a mapping between most MOF language constructs and their corresponding MetaGME constructs. Table B.1 provides an overview of this mapping. Note that MOF Classes and non-aggregate Associations find their counterparts in a number of different MetaGME constructs. For instance, MetaGME Atoms, Models, etc are all stereotyped UML Class instances. In these cases, user-defined MOF Tags are used to guide the transformation such that each MOF object maps into the proper MetaGME construct. MOF Exceptions, StructureTypes, Operations, Aliases, and EnumerationTypes which do not define the type of an Attribute have no counterparts in MetaGME. So that this information is not lost during the transformation, MOF2MetaGME encodes it as text-valued entries in the GME Registry. The BONExtender, a GME meta-level translator which generates domain-specific model interpreter C++ APIs from metamodels, can recover the encoded information and incorporate it into generated code.

MOF Construct	**MetaGME Construct**
Top-level Package	SheetFolder+ParadigmSheet
Nested Package	ParadigmSheet
Class	FCO, Atom, Model, Set, or Reference
Non-Aggregate Association	Connection, SetMembership, or ReferTo
Aggregate Association	Containment
Boolean Attribute	BooleanAttribute
Integer Attribute	Integer FieldAttribute
Double Attribute	Double FieldAttribute
String Attribute	String FieldAttribute
MOF Constraint	MetaGME Constraint
Exception, StructureType, Operation, EnumerationType, or Alias	GME Registry Node

Table B.1: MOF Construct to MetaGME Construct Mapping

Basics of GReAT

The Graph Rewriting And Transformation language (GReAT) is a model-to-model transformation language developed at Vanderbilt University[29]. GReAT supports the development of graphical language semantic translators using graph transformations. These translators can convert models of one domain into models of another domain. GReAT transformations are actually graphically-expressed transformation algorithms consisting of partially-ordered sets of primitive transformation rules. To express these algorithms, GReAT has three sub-languages: one for model instance pattern specification, one for graph transformation, and one for flow control. The GReAT execution engine takes as input a source domain metamodel, a destination domain metamodel, a set of mapping rules, and an input domain model, and then executes the mapping rules on the input domain model to generate an output domain model.

Each mapping rule is specified using model instance pattern graphs. These graphs are defined using associated instances of the modeling constructs defined in the source and destination metamodels. Each instance in a pattern graph can play one of the following three roles:

- *Bind:* Match objects in the graph.
- *Delete:* Match objects in the graph and then delete them from the graph.
- *New:* Create new objects provided all of the objects marked Bind or Delete in the pattern graph match successfully.

The execution of a primitive rule involves matching each of its constituent pattern objects having the roles Bind or Delete with objects in the input and output domain model. If the pattern matching is successful, then for each match the pattern objects marked Delete are deleted and then the objects marked New are created. The execution of a rule can also be constrained or augmented by Guards and AttributeMappings which are specified using a textual scripting language.

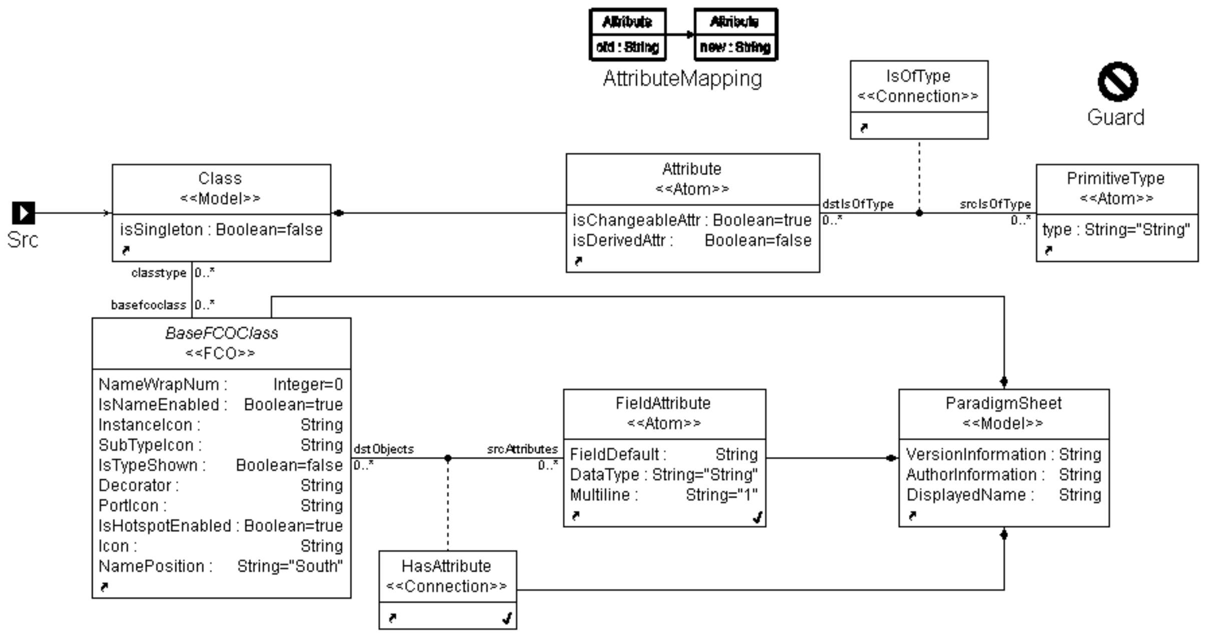

Figure B.1: MOF Primitive-Typed Attributes Mapped to MetaGME FieldAttributes

GReAT's third sub-language governs control flow. During execution, the flow of control can change from one potentially-executable rule to another based on the patterns matched (or not matched) in a rule. Flow control allows for conditional processing of input graphs. Furthermore, a graph transformation's efficiency may be increased by passing bindings from one rule to another along input and output ports to lessen the search space on a graph.

An example transformation rule is provided in Figure B.1. This figure displays the MOF2MetaGME transformation rule responsible for mapping String-, Integer-, and Double-typed MOF Attributes into MetaGME Field Attributes. The black Classes represent model patterns playing the Bind role, and the blue Classes are those which play the New role[1]. The rule finds any MOF Class containing an Attribute with an IsOfType connection to a PrimitiveType. The guard ensures that only String, Integer, or Double PrimitiveTypes are matched. If such a Class exists, then the rule finds the corresponding MetaGME Class and gives it a Field Attribute of the same type as the matched MOF Attribute.

[1]For those viewing this thesis without the benefit of color, the blue Classes are HasAttribute and FieldAttribute, while the rest of the Classes are depicted in black.

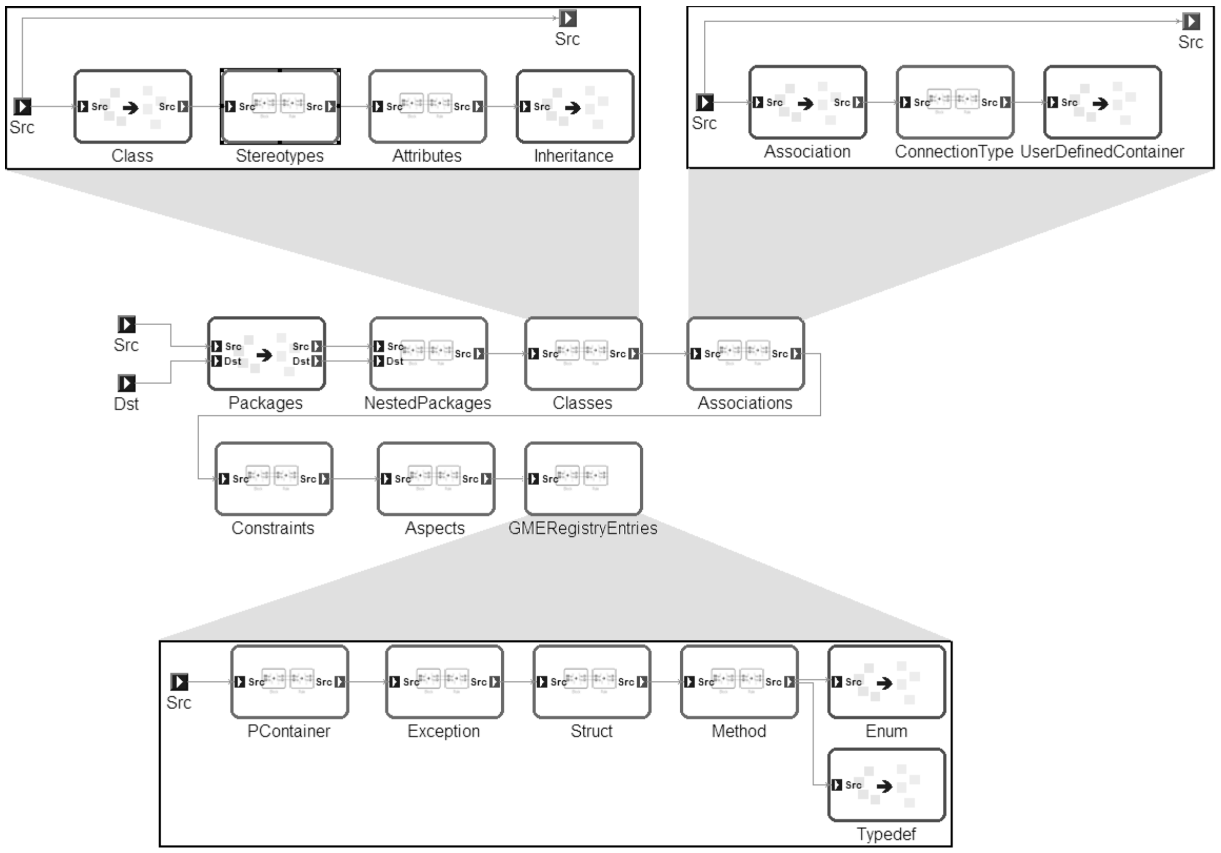

Figure B.2: MOF2MetaGME

MOF2MetaGME Implementation Overview

This section outlines the MOF2MetaGME transformation algorithm at the block-and-rule level. Figure B.2 depicts the overarching structure of MOF2MetaGME.

Packages:

For each top-level MOF Package *P*, generate a new SheetFolder containing a new ParadigmSheet both having the same name as *P*. Output all the top-level Packages and their corresponding SheetFolders.

NestedPackages:

For each MOF NestedPackage *NP*, generate a new ParadigmSheet with the same name as *NP* and contained in the Folder corresponding to the top-level Package which contains *NP*. Output all of the Packages (top-level and nested) in the MOF project.

Class:

Find and output all of the Classes contained in the various Packages of the MOF project.

Stereotypes:

Map each input MOF Class *C* into either a Model, Atom, FCO, Set, or Reference object *O* depending on the value of its GMEStereotype Tag. Contain *O* within the ParadigmSheet corresponding to the MOF Package which contains *C*. Then, for each ClassProxy *CP* which refers to *C*, generate a proxy object *PO* which refers to *O*. Contain *PO* within the ParadigmSheet corresponding to the MOF Package which contains *CP*.

Attributes:

For each Boolean-, Double-, Integer-, or String-typed Attribute owned by a Class C, respectively generate either a BooleanAttribute, Double FieldAttribute, Integer FieldAttribute, or String FieldAttribute owned by the MetaGME object corresponding to *C*.

Inheritance:

For each pair of MOF Classes *Base* and *Derived* such that *Derived* inherits from *Base*, generate an inheritance relationship such that the MetaGME object corresponding to *Derived* inherits from the MetaGME object corresponding to *Base*.

Association:

Find and output all of the Associations contained in the various Packages of the MOF project.

ConnectionType:

Find the pair of MOF Classes *Src* and *Dst* which respectively define the source type and the destination type of input Association *A*. If *A* expresses composition, generate a Containment connection from the MetaGME object corresponding to *Src* to the MetaGME object corresponding to *Dst*. Otherwise, examine the GMEConnType Tag attached to *A* generate as appropriate either a SetMembership connection, ReferTo connection, or Connection pattern from the MetaGME object corresponding to *Src* to the MetaGME object corresponding to *Dst*.

UserDefinedContainer:

This rule only executes for MOF Associations which map into MetaGME Connection patterns. Generate a Containment connection from the Connection corresponding to input MOF Association A to Model with name equal to the value of A's attached AssocClassContainer Tag.

Constraints:

For each MOF Constraint C constraining MOF model element emphE, generate a Constraint attached contained by the MetaGME object corresponding to emphE and contained in the ParadigmSheet corresponding to the Package which contains C.

Aspects:

For each Aspect defined in the MOF project, generate a same-named Aspect in the MetaGME project. The MetaGME objects visualized in the generated Aspect correspond to the MOF Classes and Associations assigned to the corresponding MOF Aspect. Note that Aspects are not a native MOF construct; however, Aspect membership may be represented through the use of MOF Tags.

PContainer:

Find and output all of the Packages and Classes of the MOF project.

Exception:

For each MOF Exception contained by some MOF Package or Class, generate a MOFException registry node in the corresponding MetaGME ParadigmSheet or stereotyped object. The value of this registry node encodes the fields (Parameters) of the Exception. These Exceptions may be thrown by MOF Operations, and map into model interpreter C++ exception classes.

Struct:

For each MOF StructureType contained by some MOF Package or Class, generate a MOF-Structure registry node in the corresponding MetaGME ParadigmSheet or stereotyped object. Set the value of this registry node to a C++ struct declaration which captures the fields of the MOF StructureType.

Method:

For each MOF Operation contained by some MOF Class, generate a MOFOperation registry node in the corresponding MetaGME object. Set the value of this registry node to a C++ method declaration which captures the parameters and return type of the MOF Operation.

Enum:

For each MOF EnumerationType contained by some MOF Package or Class and not acting as the type of any MOF Attribute, generate a MOFEnumeration registry node in the corresponding MetaGME ParadigmSheet or stereotyped object. Set the value of this registry node to a C++ enum declaration which captures the labels of the MOF EnumerationType.

Typedef:

For each MOF AliasType contained by some MOF Package or Class, generate a MOFAlias registry node in the corresponding MetaGME ParadigmSheet or stereotyped object. Set the value of this registry node to a C++ typdef declaration which aliases the type corresponding to the MOF aliased type.

BIBLIOGRAPHY

[1] Agrawal A., Karsai G., and Ledeczi A. An end-to-end domain-driven development framework. In *Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[2] Ledeczi A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., and Sprinkle J. Composing domain-specific design environments. *IEEE Computer Magazine*, pages 44–51, November 1997.

[3] Ledeczi A., Nordstrom G., Karsai G., Volgyesi P., and Maroti M. On Metamodel Composition. In *IEEE CCA*, Mexico City, May 2001.

[4] Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., and Volgyesi P. The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing*, Budapest, Hungary, May 2001.

[5] Ledeczi A., Maroti M., Karsai G., and Nordstrom G. Metaprogrammable Toolkit for Model-Integrated Computing. In *Engineering of Computer Based Systems (ECBS)*, pages 311–317, Nashville, TN, March 1999.

[6] MetaCase Consulting. *Domain-Specific Modeling: 10 Times Faster Than UML.* Available from: `www.metacase.com/papers/index.html`.

[7] Frankel D. *Model Driven Architecture.* OMG Press, 2003.

[8] Schmidt D., Gokhale A., Natarajan B., Neema S., Bapty T., Parsons J., Gray J., Nechypurenko A., and Wan N. CoSMIC: An MDA Generative Tool for Distributed Real-time and Embedded Component Middleware and Applications. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2002.

[9] Long E., Misra A., and Sztipanovits J. Increasing Productivity at Saturn. *IEEE Computer Magazine*, 1998.

[10] Institute for Software Integrated Systems. *GME 4 Users Manual Version 4.0*, 2004. Available from: `www.omg.org/docs/ad/00-09-02.pdf`.

[11] Karsai G., Agrawal A., and Ledeczi A. A Metamodel-Driven MDA Process and its Tools. In *WISME, UML 2003 Conference*, Sanfrancisco, California, October 2003.

[12] Karsai G., Nordstrom G., Ledeczi A., and Sztipanovits J. Towards Two-Level Formal Modeling of Computer-Based Systems. *Journal of Universal Computer Science*, 6(11):1131–1144, 2000.

[13] Nordstrom G., Sztipanovits J., and Karsai G. Metalevel Extension of the Multi-Graph Architecture. In *Engineering of Computer Based Systems (ECBS)*, pages 61–68, Jerusalem, Israel, April 1998.

[14] Nordstrom G., Sztipanovits J., Karsai G., and Ledeczi A. Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments. In *Proceedings of the IEEE ECBS'99 Conference*, pages 68–74, Nashville, TN, April 1999.

[15] Object Management Group. Model Driven Architecture. Available from: `www.omg.org/mda`.

[16] Object Management Group. UML. Available from: `www.omg.org/uml`.

[17] Object Management Group. *Meta Object Facility Specification v1.4*, 2002. Available from: `www.omg.org/docs/formal/02-04-03.pdf`.

[18] Object Management Group. *Meta Object Facility Specification v2.0*, 2002. Available from: `www.omg.org/cgi-bin/apps/doc?ptc/03-10-04.pdf`.

[19] Object Management Group. *Common Warehouse Metamodel (CWM) Specification*, 2003. Available from: `www.omg.org/docs/formal/03-03-02.pdf`.

[20] Object Management Group. *UML 2.0 OCL Specification*, 2003. Available from: `www.omg.org/docs/ptc/03-10-14.pdf`.

[21] Object Management Group. *Unified Modeling Language: Superstructure version 2.0, 3rd revised submission to OMG RFP*, 2003. Available from: `www.omg.org/docs/ad/00-09-02.pdf`.

[22] Object Management Group. *XML Metadata Interchange (XMI) Specification v2.0*, 2003. Available from: `www.omg.org/docs/formal/03-05-02.pdf`.

[23] Object Management Group. *UML Profile for Meta Object Facility, v1.0*, 2004. Available from: `www.omg.org/docs/formal/04-02-06.pdf`.

[24] Honeywell International Inc. *DOME Users Guide.* Available from: `www.htc.honeywell.com/dome/support.htm#documentation`.

[25] Sprinkle J., Karsai G., Ledeczi A., and Nordstrom G. The New Metamodeling Generation. In *Engineering of Computer Based Systems (ECBS)*, page 275, Washington, DC, April 2001.

[26] Sztipanovits J. and Karsai G. Model-Integrated Computing. *IEEE Computer Magazine*, pages 110–112, April 1997.

[27] Sztipanovits J., Karsai G., Biegl C., Bapty T., Ledeczi A., and Malloy D. MULTIGRAPH: An Architecture for Model-Integrated Computing. In *Proceedings of the International Conference on Engineering of Complex Computer Systems*, pages 361–368, Ft. Lauderdale, FL, November 1995.

[28] G. Karsai. A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming. *IEEE Computer Magazine*, pages 36–44, March 1995.

[29] G. Karsai, A. Agrawal, and F. Shi. On the use of graph transformations for the formal specification of model interpreters. *Journal of Universal Computer Science*, 9(11):12961321, 2003.

[30] LabView. Available from: `www.ni.com/labview`.

[31] Emerson M. and Sztipanovits J. Implementing a MOF-Based Metamodeling Environment Using Graph Transformations. In *4th OOPSLA Workshop on Domain-Specific Modeling*, pages 83–92, Vancouver, Canada, October 2004.

[32] Emerson M., Sztipanovits J., and Bapty T. A MOF-Based Metamodeling Environment. *Journal of Universal Computer Science*, 10(10):1357–1382, October 2004.

[33] Matlab. Available from: `www.mathworks.com`.

[34] Neema S. and Karsai G. Embedded Control Systems Language for Distributed Processing. Technical Report ISIS-04-505, Institute for Software Integrated Systems, May 2004.

[35] Neema S., Sztipanovits J., and Karsai G. Design-Space Construction and Exploration in Platform-Based Design. Technical Report ISIS-02-301, Institute for Software Integrated Systems, 2002.

[36] Clark T., Evans A., Sammut P., and Willans J. *Applied Metamodelling: A Foundation for Language Driven Development v 0.1.* Xactium, 2004.

[37] Clark T., Evans A., Sammut P., and Willans J. An eXecutable Metamodeling Facility for Domain Specific Language Design. In *4th OOPSLA Workshop on Domain-Specific Modeling*, pages 103–109, Vancouver, Canada, October 2004.

[38] Clark T., Evans A., Kent S., and Sammut P. The MMF Approach to Engineering Object-Oriented Design Languages. In *Workshop on Language Descriptions, Tools and Applications*, 2001.