Journal of Universal Computer Science, vol. 10, no. 10 (2004), 1357-1382
submitted: 30/6/04, accepted: 6/10/04, appeared: 28/10/04 © J.UCS

A MOF-Based Metamodeling Environment

Matthew J. Emerson
(Vanderbilt University, USA
Institute for Software Integrated Systems (ISIS)
mjemerson@isis.vanderbilt.edu)

Janos Sztipanovits
(Vanderbilt University, USA
Institute for Software Integrated Systems (ISIS)
sztipaj@isis.vanderbilt.edu)

Ted Bapty
(Vanderbilt University, USA
Institute for Software Integrated Systems (ISIS)
bapty@isis.vanderbilt.edu)

Abstract: The Meta Object Facility (MOF) forms one of the core standards of the Object Management Group's Model Driven Architecture. It has several use-cases, including as a repository service for storing abstract models used in distributed object-oriented software development, a development environment for generating CORBA IDL, and a metamodeling language for the rapid specification, construction, and management of domain-specific technology-neutral modeling languages. This paper will focus on the use of MOF as a metamodeling language and describe our latest work on changing the MIC metamodeling environment from UML/OCL to MOF. We have implemented a functional graphical metamodeling environment based on the MOF v1.4 standard using GME and GReAT. This implementation serves as a testament to the power of formally well-defined metamodeling and metamodel-based model transformation approaches.Furthermore, our work gave us an opportunity to evaluate sevaral important features of MOF v1.4 as a metamodeling language:

- Completeness of MOF v1.4 for defining the abstract syntax for complex (multiple aspect) DSML-s
- The Package concept for composing and reusing metamodels
- Facilities for modeling the mapping between the abstract and concrete syntax of DSML-s

Key Words: Model Driven Architecture, Model-Integrated Computing, graph transformations
Category: D.2.2 Tools and Techniques

1 Introduction

Model-Integrated Computing (MIC) [Sztipanovits and Karsai 1997] is a comprehensive approach to model-based design directed toward embedded software and system development. In embedded computing, where the role of the embedded software is to "configure" the computing device so as to meet physical requirements, it is not surprising that when using current software technology *physical properties are not composable* — rather, they appear as cross-cutting constraints in the development process. Consequently, in MIC we had to go beyond conventional software technology to a model-based system design technology which addresses the design of the whole system with its many interdependent physical, functional and logical aspects.

During the past years, model-based design has become one of the major trends in software and systems engineering. Model Driven Architecture (MDA), the central vision of the OMG, aims to provide a platform-independent approach to domain-specific application development and promotes the creation of software systems through modeling and model transformation [OMG 2002]. MDA is a logical continuation of OMG's successful introduction of UML, which originally provided a common visual notation for object-oriented design. The MDA vision extends the use of modeling to all stages of the software development process. Similarly to MIC, MDA considers the software development process as a sequence of transformations among models.

In spite of the similarities of the MIC and MDA visions, there has been one strong difference: the role of Domain Specific Modeling Languages (DSML). In MIC, the use of DSML-s is not an option; it is mandatory. In embedded systems, models capturing only the logical characteristic of applications are not sufficient to make the physical properties computable and analyzable. The models must take into account the physical properties of the platforms and the embedding environment. Therefore, the scope of modeling and the level of abstraction required is highly domain-specific. We cannot expect that the same kinds of models and modeling languages which are used in the design of 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).

In MDA, the prevailing view is that UML will be *the* single, universal, platform independent modeling language from which model translators will generate software artifacts on specific platforms. This strongly-held conviction has its roots in viewing model-based design pretty much the same way as conventional programming, where language standardization has been a vital issue. The problem is that the scope of model-based design is much broader than programming. Model-based design is built around the modeling process, which inherently includes the selection of essential aspects, careful separation of the modeled and not modeled worlds, and abstraction. Unless one believes that a universal language can be created which is broad enough to cover all conceivable systems, reasonable solutions require DSML-s.

A half-step in the right direction is to use a "universal" modeling language, but to make it extendable. This approach was reflected by the UML profile mechanism. UML profiles are stereotyped packages that contain model elements extended with stereotypes, tagged values and constraints [Nordstrom 1999]. However, stereotyping does not change fundamental syntactic and semantic properties of the modeling languages and tends to create a complex web of interfering standards.

A more radical approach to constructing DSML-s is based on understanding the fundamentals of constructing modeling languages and creating standards and tool suites for facilitating their specification and composition. The core concept in this approach is metamodeling. Metamodels are models of DSML-s expressed in specific metamodeling languages. In MIC we have developed and successfully applied the metamodeling approach in a variety of application domains [Long et al. 1998], [Neema et al. 2002]. The latest developments in UML 2 [OMG 2003] also shift more attention to this approach. UML 2 has been defined using a standard metamodeling language, the Meta Object Facility (MOF), and one of the MOF use-cases [OMG 2002] is the specification of DSML-s.

The goal of this paper is to describe our latest work on changing the MIC metamodeling environment from UML/OCL to MOF. The work gave us an opportunity to evaluate MOF as a metamodeling language, particularly in terms of its support for DSML composition. Our implementation of the MOF-based metamodeling environment (GME-MOF) used the meta-programmable Generic Modeling Environment (GME) [Ledeczi, Bakay et al. 2001], a core tool of the MIC technology. We believe that this implementation serves as a testament to the power of formally well-defined metamodeling and metamodel-based model transformation approaches. First, we will provide a short summary of the formal specification of DSML-s. This summary will be followed by an overview and evaluation of MOF as a metamodeling language. The last section of the paper describes the implementation of GME-MOF using metamodeling and model transformations.

2 MODELING AND COMPOSITION OF DSML-s

Formally, a DSML is a five-tuple of concrete syntax (C), abstract syntax (A), semantic domain (S) and semantic and syntactic mappings (MS, and MC) [Clark et al. 2001]:

$$L = < C, A, S, MS, MC >$$

The concrete syntax C defines the specific notation used to express models, which may be graphical, textual or mixed. The abstract syntax A defines the concepts, relationships, and integrity constraints available in the language. The semantic domain S is usually defined by means of some mathematical formalism in terms of which the meaning of the models is explained. The syntactic mapping $MC : A \rightarrow C$ assigns syntactic constructs (graphical, textual or both) to the elements of the abstract syntax. The semantic mapping $MS : A \rightarrow S$ relates syntactic concepts to those of the semantic domain. Any DSML which is to be used in the development process of embedded systems requires the precise specification (or modeling) of all five components of the language definition. The languages which are used for defining components of DSML-s are called meta-languages and the concrete, formal specifications of DSML-s are called metamodels [OMG 2002].

The specification of the abstract syntax of DSML-s requires a meta-language that can express concepts, relationships, and integrity constraints. In MIC, we adopted UML class diagrams and the Object Constraint Language (OCL) as our meta-language. This selection is consistent with UML's and MOF's four layer meta-modeling architecture (see e.g. [OMG 2003]). The semantic domain and semantic mapping defines semantics for a DSML, and these semantics give a precise meaning to those models that we can create using the modeling language. Naturally, models might have different interesting properties; therefore a DSML might have a multitude of semantic domains and semantic mappings associated with it. For example, structural and behavioral semantics are frequently associated with DSML-s. The structural semantics of a modeling language 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 by some form of set-valued semantics. The behavioral semantics describes the evolution of the state of the modeled artifact along some time model. 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 $MC : A \rightarrow C$) since they play the key role in tools and model transformations. Issues related to modeling semantics are discussed elsewhere (e.g. [Clark et al. 2001]).

2.1 Metamodeling with GME

GME is a metaprogrammable model builder for creating domain-specific modeling environments and then modeling in those environments. It provides a graphical metamodeling language called MetaGME for the specification of DSML-s. MetaGME can specify the concrete syntax and abstract syntax of a target graphical modeling language. GME-based metamodeling is shown in Figure 1.

The metamodel ${}_{MetaGME}MM_{DSML}$ of a DSML captures the abstract syntax ${}_{MetaGME}A_{DSML}$, concrete syntax ${}_{MetaGME}C_{DSML}$, and syntactic map-

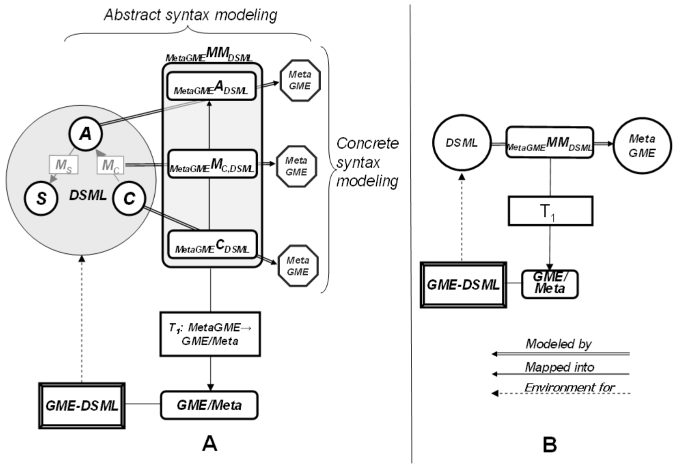

Figure 1: A) Metamodeling with GME **B)** Simplified Diagram

ping ${}_{MetaGME}MC_{DSML}$ using the constructs of MetaGME. MetaGME is essentially an extension of UML Class Diagrams with OCL constraints that uses the Class stereotype facility to imply the abstract syntax expressed by the metamodel [Nordstrom 1999]. Because we have implemented our MOF-based metamodeling environment using MetaGME, a basic understanding of some of these stereotypes is necessary to understand our implementation. The meanings of the MetaGME stereotypes used in the GME-MOF model are as follows:

- *Models* are compound objects which are visualized in GME as containing other model elements.
- *Atoms* are elementary objects which are not visualized in GME as containing other model elements.
- *FCO-s* are first-class objects which must be abstract but can serve as the base type of an element of any other stereotype.
- *References* correspond to pointers in an object-oriented programming language.
- *Connections* are analogous to UML Association Classes.
- *Aspects* provide logical visibility partitioning to present different views of a model.

The ${}_{MetaGME}MM_{DSML}$ metamodel is translated by the T_1 translator (called the meta-interpreter) into a GME/Meta configuration file for GME. Using this configuration file, GME will serve as the domain-specific modeling environment for the target domain. (A simplified diagram of the metamodeling process can be seen in Figure 1.B)

2.2 Experience with Metamodeling

We have extensively used MetaGME for modeling DSML-s. Based on our experience, 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

3 MOF Overview

In this section, we describe the MOF model and explore how well MOF meets each of these concerns.

3.1 The MOF Architecture

MOF's architecture conforms to the classic four-metalayer metamodeling framework where each level consists of instances of elements of the next higher level:

- *M0 Level:* The concrete data of the system of interest at some point in time
- *M1 Level:* The model of the organization and behavior of a system using domain-specific concepts
- *M2 Level:* The metamodel or domain-specific modeling language capable of expressing the structure and semantics of the system metadata (UML is the classic example)
- *M3 Level:* MOF, a self-describing meta-metamodel for specifying domain-specific modeling languages

Further metalayers beyond M3 to specify MOF are not necessary because MOF is self-describing (or metacircular). In essence, a metamodeling language such as MOF is simply a DSML for modeling metamodels; as a specifier of such languages, MOF is fully described using its own modeling concepts.

3.2 Basic MOF Concepts

As defined in the v1.4 specification [OMG 2002], MOF provides the following five basic object-oriented concepts for use in defining metamodels:

- *Classes* are types whose instances have identity, state, and behavior. The state of a Class is expressed by its Attributes and Constants, and its behavior is governed by Operations and Exceptions. Constraints can place limitations on both the state and the behavior of a Class.
- *Associations* describe binary relationships between Classes. They may express composite or non-composite aggregation semantics. Because MOF Associations have no object identity, they lack both state and behavior. This deficiency makes the specification of some metamodels more awkward and difficult.
- *DataTypes* are types with no object identity. By design the different MOF DataTypes encompass most of the CORBA IDL primitive and constructed types.
- *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.
- *Constraints* specify the well-formedness rules that govern valid domain models.

MOF provides several features for metamodel composition, extension, and reuse, including 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, Classes can only inherit from other Classes and Packages can only inherit from other Packages.

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. It 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 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 metamodel described by an importing Package cannot be used to create instances of the Classes of the metamodel described by the imported Package. The importing Package can however make use of each of the Classes of the imported Package as if it had acquired them through Class importation, create DataTypes and Constants using imported DataTypes, and define Operations which raise imported Exceptions.

4 Shortcoming of MOF

We explore three aspects of MOF that prevent it from being an ideal language for the specification of graphical DSML-s, including its lack of a natural way to specify the concrete syntax of DSML-s, its lack of a way to specify different logical views for separately visualizing relevant aspects of a domain model, and its lack of stateful Associations.

4.1 Specifying DSML Concrete Syntax

The UML Profile for Meta Object Facility v1.0 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 [OMG 2004]. This mapping is needed because the MOF specification itself does not provide a concrete syntax for MOF. Moreover, MOF actually lacks any natural facility for specifying the concrete syntax of any metamodel. This is an important point because we are interested in using MOF as a graphical facility for the specification of graphical DSML-s. Consider the simple MOF metamodel for finite state machines in Figure 2, which employs the UML-like syntax recommended by [OMG 2004]. What do State instances look like? Or the Transition instances between State instances? There is no easy way to address these concerns using MOF.

4.2 Specifying Multiple System Views

Although MOF provides the concept of nested packages for logically partitioning the namespace of a metamodel, it provides no way to specify different logical

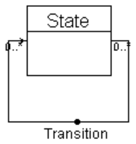

Figure 2: Finite State Machine Metamodel

views of domain models. Consequently, modelers are unable to visually separate the different interacting concerns of the system.

4.3 Including State in Associations

MOF's lack of support for Associations with state makes the definition of some DSML-s awkward. For example, consider again our example finite state machine metamodel above (Figure 2). Finite state machines already have a well-defined graphical syntax: States are represented as the nodes of a graph and directed arcs between the nodes represent Transitions between states. Each arc is labeled with a letter of the input alphabet. But how do we model the fact that every Transition is associated with a label in MOF?

If MOF Associations had state, we could 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 will need to 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 3. Now, when a user wants to model a specific finite state machine, 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 DSML-s similarly require stateful Associations, and the inclusion of extra Classes to carry the burden of this state seems unnecessary — MOF should simply include stateful Associations.

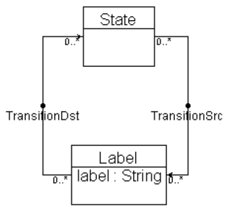

Figure 3: Finite State Machine Metamodel with Labels

5 GME-MOF

GME-MOF is our graphical implementation of the MOF v1.4 for the specification of DSML-s in the Generic Modeling Environment, one of the flagship products following model-based approach to system design. We have provided a graphical interface along the lines of UML Class Diagrams for GME-MOF and we have extended the MOF to make up for some of the shortcomings discussed in the previous section: GME-MOF can specify the concrete syntax of graphical DSML-s and it includes the concept of Aspects to present different logical views of a domain model. An overview of our GME-MOF implementation is shown in Figure 4.

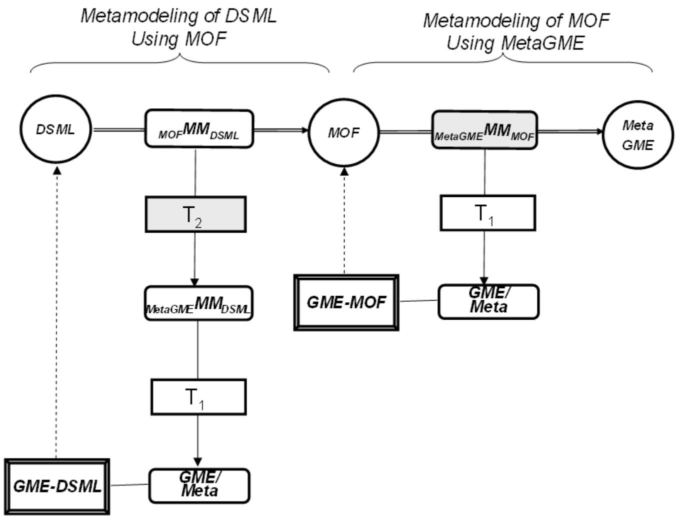

Figure 4: Building the MOF-Based Metamodeling Environment

As described previously, any complete GME metamodeling language must provided two facilities: a graphical metamodeling environment for the specification of the abstract syntax, concrete syntax, and syntactic mappings, and a translation tool capable of generating from the graphical metamodel of a target domain the configuration file that enables GME to serve as the domain-specific modeling environment for that domain. Because MetaGME is itself a metamodeling language, it has sufficient expressive power to fully describe MOF. So, while MOF specification uses MOF to describe itself, we have described MOF in the language of MetaGME. Furthermore, because MetaGME already reflects the full range of configurations realizable in GME (in fact, the meta-information in the configuration files directly parallels the MetaGME modeling concepts), we found the easiest way to create the necessary translation tool by defining a transformation algorithm from MOF-specified metamodels into MetaGME-specified metamodels. Then, we take advantage of MetaGME's existing meta-interpreter to generate the GME configuration files. The shaded components in Figure 4 represent the facilities we have to provide in implementing MOF for GME: ${}_{MetaGME}MM_{MOF}$ is our MetaGME-specified MOF model and T_2 is our implementation of the transformation algorithm from MOF-specified metamodels to MetaGME-specified metamodels. T_1 is MetaGME's meta-interpreter, which generates the GME configuration files from the translated metamodels.

The transformation algorithm is quite straightforward, because the concepts of both MOF and MetaGME are rooted in the concepts of UML Class Diagrams. In order to facilitate this transformation, we have implied a natural mapping between MetaGME and MOF concepts in our implementation of the MOF metamodeling environment and extended the appropriate MOF model elements with attributes to specify the concrete syntax and some elements of the abstract syntax of DSML-s. The natural choice for implementing our metamodel translation algorithm is the Graph Rewriting and Transformation toolsuite, GReAT [Agrawal et al. 2003]. GReAT is a DSML implemented for GME that enables the graphical specification of graph transformations with a formal execution semantics [Karsai et al. 2003]. Because both MetaGME and the GME-MOF metamodels are visually represented as annotated graphs, we can describe the translation from one kind of metamodel to the other using the language of graph transformations. Furthermore, using GReAT carries all the natural benefits of using a DSML — we were able to easily and rapidly create our metamodel translation tool and the tool itself is easy to analyze, maintain, and evolve. We have named this tool MOF2MetaGME. We describe the implementation of GME-MOF in Appendix A and the implementation of MOF2MetaGME in Appendix B.

To illustrate the function of MOF2MetaGME, we provide figures Figures 5 and 6. Figure 5 is a small part of a GME-MOF implementation of UML Class Diagrams for GME used as the input to MOF2MetaGME, and Figure 6 is the corresponding output from MOF2MetaGME. Note the high degree of symmetry between the two diagrams. The ClassCopy and ClassDiagram Classes present in the MOF-specified metamodel are necessary to make use of GME's facilities for multi-sheet modeling and visual hierarchy, respectively. Because we are mapping MOF-metamodels onto MetaGME-metamodels, there is an extent to which we have to keep our target platform — GME — in mind during metamodel development.

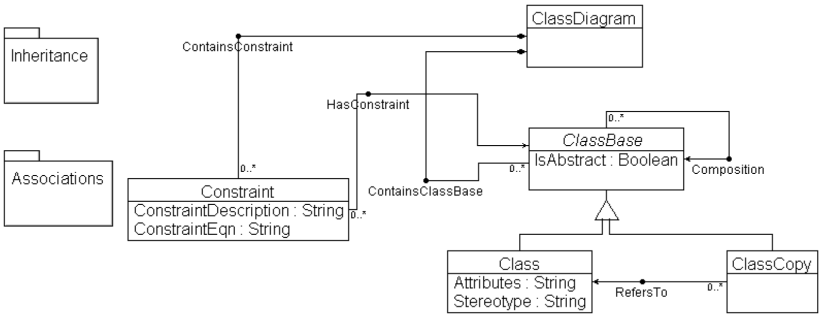

Figure 5: UML Class Diagrams in MOF

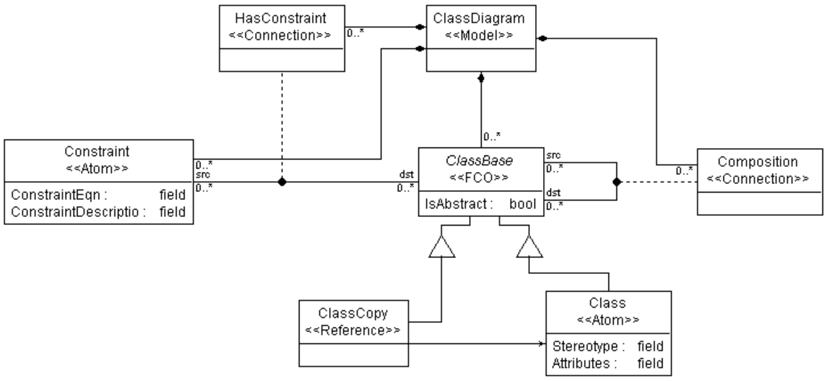

Figure 6: UML Class Diagrams in MetaGME

6 Discussion

An important component of our vision for the future of MIC involves the development of libraries of canonical metamodels which may be extended and composed to expedite the development of complex DSML-s. Consequently, we see a demand for a metamodeling language with powerful, expressive features supporting metamodel composition. We find the metamodel composition features offered by MOF v1.4, Package importation and generalization, to be too rudimentary to support our vision. Primarily, MOF lacks any facility for specifying "join points" between metamodels — model elements with potentially different names, state, behavior, and constraints, but which actually abstract the same concept in two different domains. Moreover, the fact that MOF restricts the names of the elements in the Packages from which a given Package inherits makes the job of creating libraries of composable metamodels more difficult — in order to insure the composability of the metamodels, we must adopt a naming convention to prevent name collisions between the model elements. Such a naming convention might obfuscate the relationship between a library metamodel and the domain it abstracts. MetaGME, on the other hand, already enables the identification of join points through the use a built-in Equivalence operator [Ledeczi, Nordrtom et al. 2001]. Furthermore, MetaGME provides more fine-grained facilities for inheritance than MOF: implementation inheritance and interface inheritance. Implementation inheritance allows a derived type to inherit only the state of the base type as well as those composition relationships in which the base type plays the role of container. Interface inheritance allows a derived type to inherit only the relationships in which the base type does not play the role of container. The union of implementation and interface inheritance is standard inheritance. We would like to see future revisions of MOF incorporate more advanced features like these for metamodel composition and reuse.

References

[Sztipanovits and Karsai 1997] Sztipanovits, J., Karsai, G.: "Model-integrated computing"; IEEE Computer Magazine (Apr 1997), 110112.

[OMG 2002] Object Management Group: "The model driven architecture, 2002"; http://www.omg.org/mda (2002).

[Nordstrom 1999] Nordstrom, G.: "Metamodeling — Rapid Design and Evolution of Domain-Specific Modeling Environments"; Proc. IEEE ECBS '99 Conference (1999).

[Long et al. 1998] Long, E., Misra, A., Sztipanovits, J.: "Increasing Productivity at Saturn"; IEEE Computer Magazine (August 1998).

[Neema et al. 2002] Neema S., Sztipanovits J., Karsai G.: "Design-Space Construction and Exploration in Platform-Based Design"; ISIS-02-301 (2002).

[OMG 2003] Object Management Group: "Unified Modeling Language: Superstructure version 2.0, 3rd revised submission to OMG RFP"; http://www.omg.org/docs/ad/00-09-02.pdf (2003).

[OMG 2002] Object Management Group: "Meta Object Facility Specification v1.4"; http://www.omg.org/docs/formal/02-04-03.pdf (2002).

[Ledeczi, Bakay et al. 2001] Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J.: "Composing domain-specific design environments"; IEEE Computer Magazine (Nov 2001), 4451.

[Clark et al. 2001] Clark, T., Evans, A., Kent, S., Sammut, P.: "The MMF Approach to Engineering Object-Oriented Design Languages"; Workshop on Language Descriptions, Tools and Applications (2001).

[Rumpe 1998] Rumpe, B.: "A Note on Semantics (with an Emphasis on UML)"; Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications) (1998).

[OMG 2004] Object Management Group: "UML Profile for Meta Object Facility, v1.0"; http://www.omg.org/docs/formal/04-02-06.pdf (2004).

[Agrawal et al. 2003] Agrawal, A., Karsai, G., Ledeczi, A.: "An end-to-end domain-driven development framework"; Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (2003).

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

[Ledeczi, Nordrtom et al. 2001] Ledeczi, A., Nordstrom, G., Karsai, G., Volgyesi, P., Maroti, M.: "On Metamodel Composition"; IEEE CCA (2001).
[ISIS 2004] ISIS: "GME 4 Users Manual Version 4.0"; Institute for Software Integrated Systems (2004).

Appendix A GME-MOF Implementation

We have provided an abbreviated specification of our MOF Model (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 [ISIS 2004].

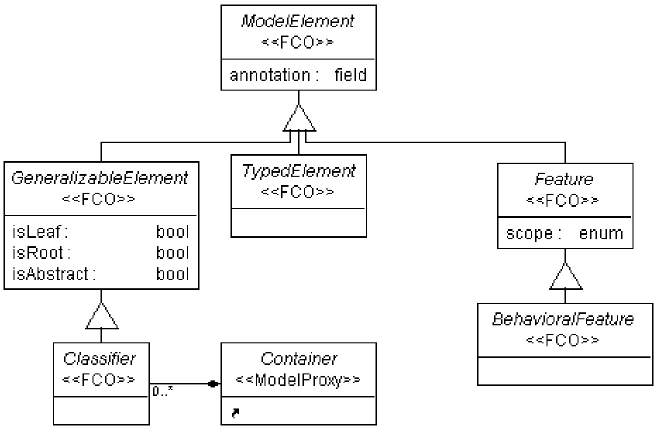

Figure 7: Abstract Base Classes

Appendix A.1 Abstract Base Classes (Figure 7)

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.

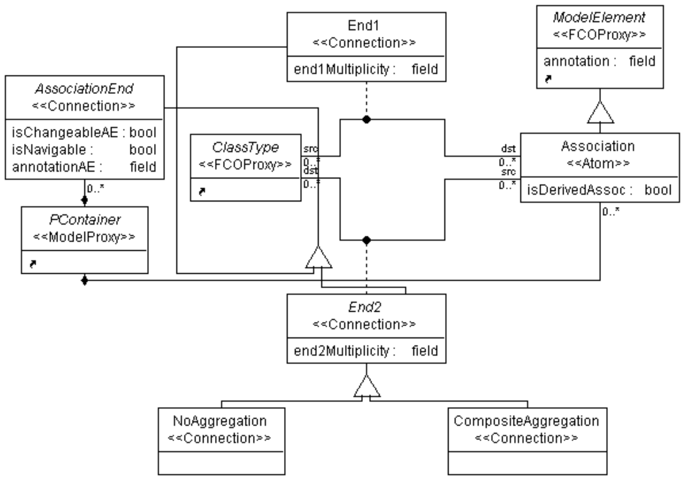

Figure 8: Association

Appendix A.2 Association (Figure 8)

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.

Appendix A.3 Class, Attribute, and Operation (Figure 9)

Constraints:
Name: NotSingletonAndAbstract
Constrains: Class
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'.

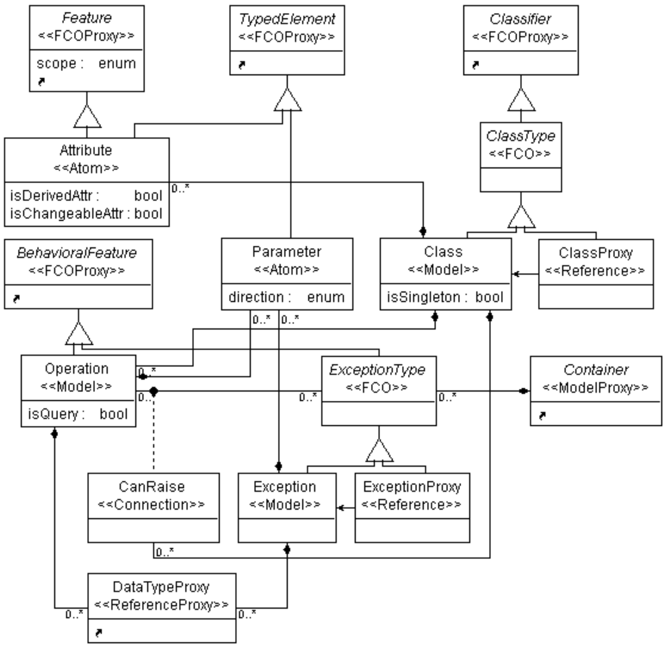

Figure 9: Class, Attribute, and Operation

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.

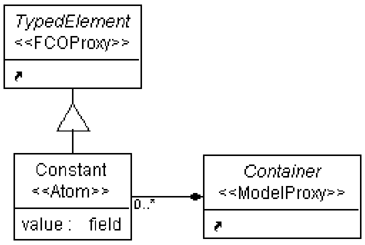

Figure 10: Constant

Appendix A.4 Constant (Figure 10)

Constraints:
Name: TypeIsPrimitive
Constrains: Constant
Description: Constants must have primitive types.

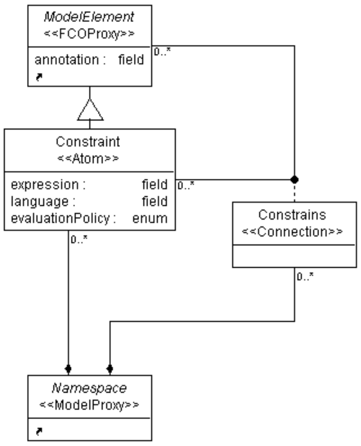

Figure 11: Constraint

Appendix A.5 Constraint (Figure 11)

Constraints:
Name: ValidElement
Constrains: 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.

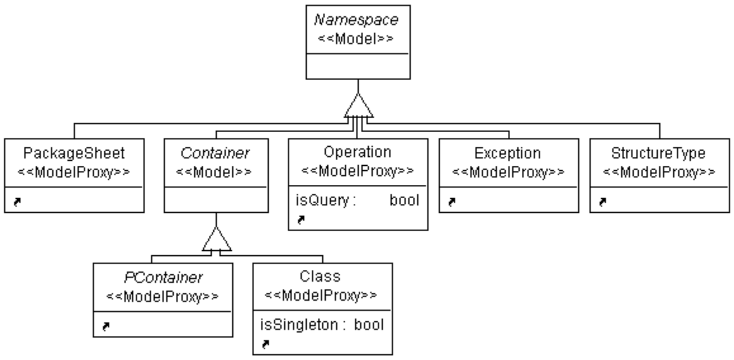

Figure 12: Containment

Appendix A.6 Containment (Figure 12)

Constraints:
Name: NoNameCollisions
Constrains: Namespace
Description: The contents of a Namespace may not collide.

Appendix A.7 DataType (Figure 13)

Constraints:
Name: NotAbstract

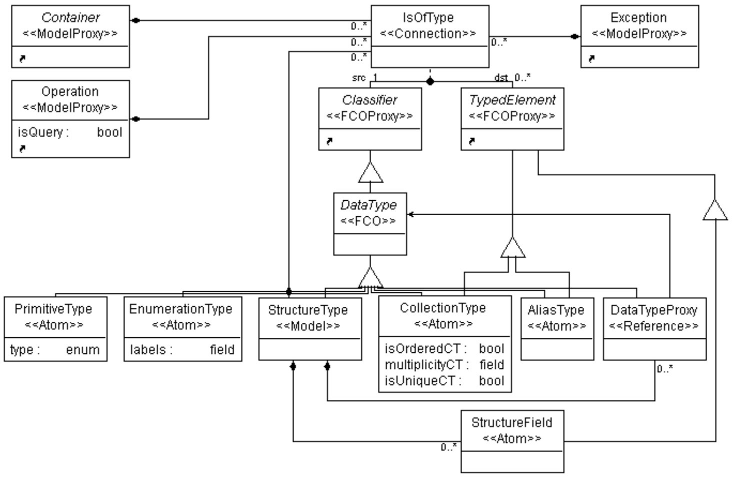

Figure 13: DataType

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.

Visualization:
DataType and IsOfType are visible in the Features Aspect.

Appendix A.8 Generalization (Figure 14)

Constraints:
Name: HasDerived
Constrains: Inheritance

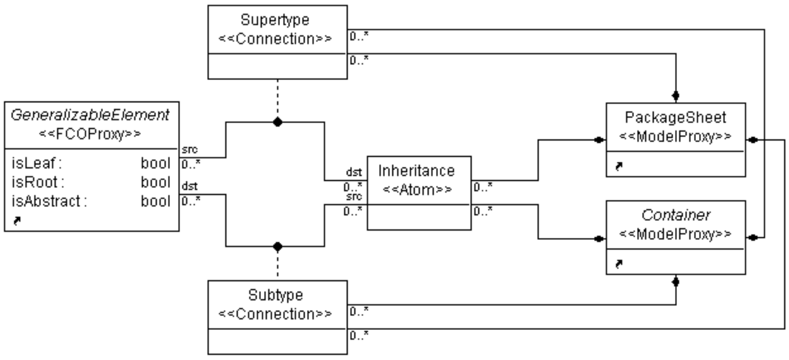

Figure 14: Generalization

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: 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: LeafCannotSpecialize
Constrains: GeneralizableElement
Description: Leaf elements cannot be specialized.

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

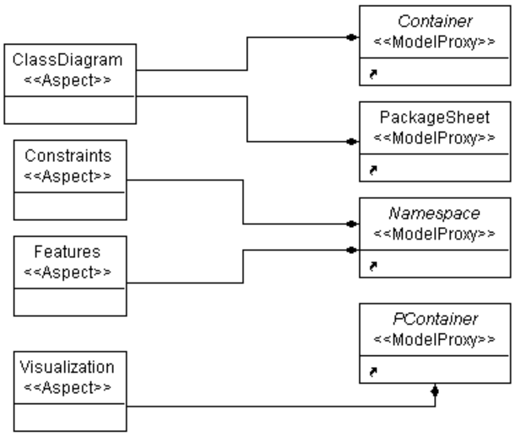

Figure 15: GME-MOF Aspects

Appendix A.9 GME-MOF Aspects (Figure 15)

No constraints, labels, or visualization information.

Appendix A.10 Package (Figure 16)

Constraints:
Name: NotAbstract
Constrains: PContainer
Description: A package may not be declared abstract.

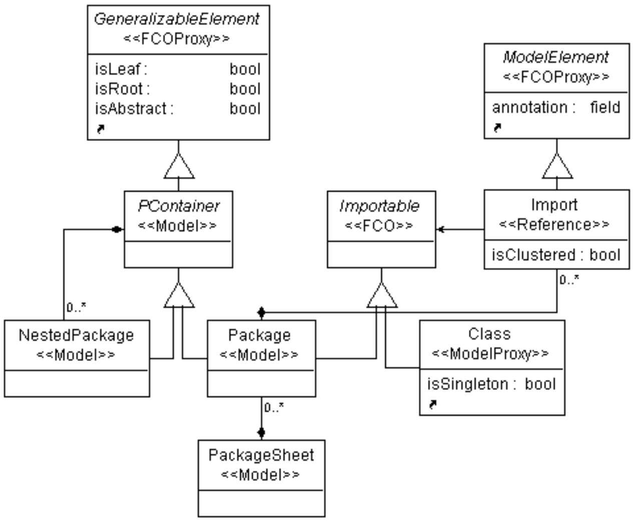

Figure 16: Package

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.

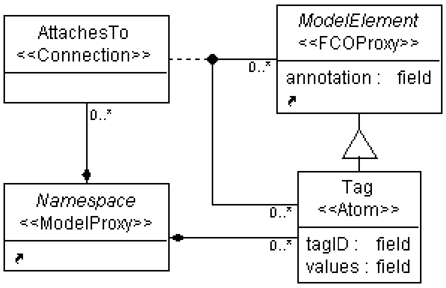

Figure 17: Tag

Appendix A.11 Tag (Figure 17)

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

Appendix A.12 Multi-Aspect Modeling (Figure 18)

Constraints:
Name: ModelsHaveAspects
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

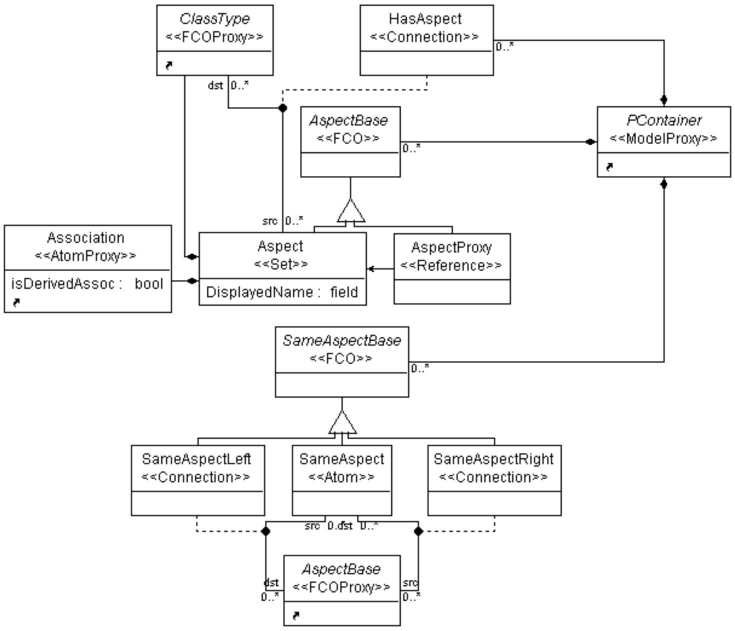

Figure 18: Multi-Aspect Modeling

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.

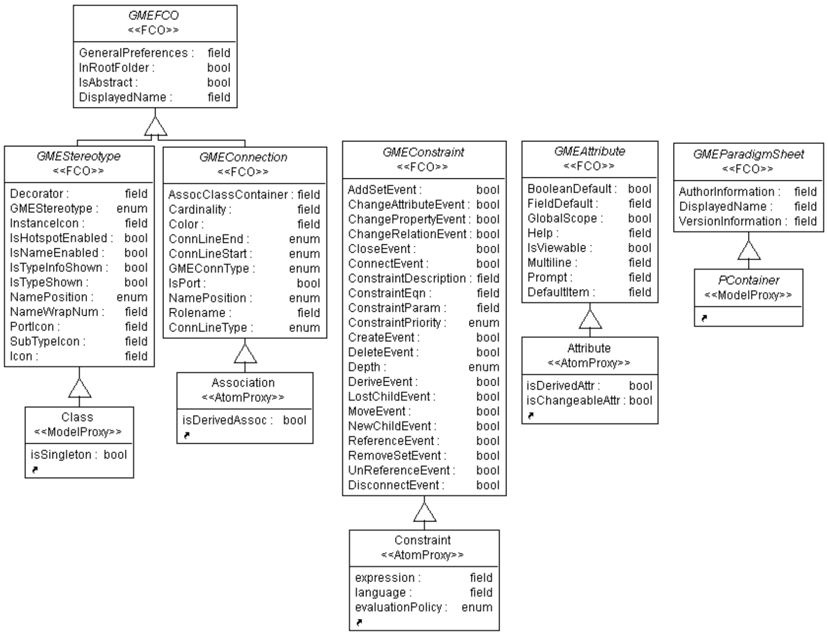

Figure 19: GME Mappings

Appendix A.13 GME Mappings (Figure 19)

No constraints, labels, or visualization information. 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 [ISIS 2004]. Note that these attributes may be conceptualized as MOF Tags applied on an element-by-element, metamodel-by-metamodel basis.

Appendix B MOF2MetaGME Implementation

MOF2MetaGME is an implementation of a graph transformation algorithm for converting GME-MOF metamodels into MetaGME metamodels. This transformation algorithm has been implemented using GReAT, the Graph Rewriting and Transformation toolsuite for GME. In order to avoid a detailed discussion of syntax and semantics of GReAT, we provide only a very high-level description of the transformation rules implemented for MOF2MetaGME. It should be noted that the transformation from GME-MOF to MetaGME is not isomorphic — MOF includes a number of concepts and capabilities that MetaGME does not, including (among others) singleton Classes, arbitrarily-typed Attributes, and derived Attributes and Associations.

Appendix B.1 Packages

Each GME-MOF Package generates a MetaGME SheetFolder and ParadigmSheet. Each GME-MOF NestedPackage generates a MetaGME ParadigmSheet.

Appendix B.2 Classes

For each GME-MOF Class generates either a MetaGME FCO, Atom, Model, Set, or Reference as dictated by the Class's GMEStereotype attribute. Each GME-MOF ClassProxy generates either a MetaGME FCOProxy, AtomProxy, ModelProxy, SetProxy, or ReferenceProxy depending on the GMEStereotype attribute of the ClassProxy's referent Class. If one GME-MOF Class inherits from another, then MetaGME stereotyped Classes generated from those GME-MOF Classes also inherit from one another.

Appendix B.3 Attributes

Each GME-MOF String-typed Attribute generates a MetaGME FieldAttribute of type String. Each GME-MOF Integer-typed Attribute generates a MetaGME FieldAttribute of type Integer. Each GME-MOF Double-typed Attribute generates a MetaGME FieldAttribute of type Double. Each GME-MOF Boolean-typed Attribute generates a MetaGME BooleanAttribute. Each GME-MOF EnumerationType-typed Attribute generates a MetaGME EnumAttribute with the same fields. GME-MOF Attributes of all other types are ignored, as there is no corresponding MetaGME representation.

Appendix B.4 Associations

Each GME-MOF Association generates a MetaGME Containment connection if the Association has composite aggregation semantics. Otherwise, it generates a ReferTo, SetMembership, or User-Defined connection depending on the Association's GMEConnType attribute.

Appendix B.5 Constraints

Each GME-MOF Constraint generates a MetaGME Constraint.

Appendix B.6 Aspects

Each GME-MOF Aspect generates a MetaGME Aspect.