Published in the Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Leung, Sang H. Son, CRC Press, 2006

Metamodeling Languages and Metaprogrammable Tools

Matthew Emerson and Sandeep Neema and Janos Sztipanovits

Institute for Software Integrated Systems

Vanderbilt University

Nashville, TN 37203

E-mail: mjemerson@isis.vanderbilt.edu

June 29, 2006

1 Introduction

The convergence of control systems and software engineering is one of the most profound trends in technology today. Control engineers build on computing and communication technology to design robust, adaptive and distributed control systems for operating plants with partially known nonlinear dynamics. Systems engineers face the problem of designing and integrating large scale systems where networked embedded computing is increasingly taking over the role of "universal system integrator". Software engineers need to design and build software that needs to satisfy requirements that are simultaneously physical and computational. This trend drives the widespread adoption of model-based design techniques for computer-based systems. The use of models on different levels of abstraction have been a fundamental approach in control and systems engineering. Lately, model-based software engineering methods, such as OMG's Model-Driven Architecture (MDA) [13][7] have gained significant momentum in the software industry, as well. The confluence of these factors has led to the emergence of model-driven engineering (MDE) that opens up the opportunity for the fusion of traditionally isolated disciplines.

Model-Integrated Computing (MIC), one practical manifestation of the general MDE vision, is a powerful model-based design approach and tool suite centered on the specification and use of semantically-rich,

Published in the Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Leung, Sang H. Son, CRC Press, 2006

domain-specific modeling languages (DSMLs) during the full system life-cycle [19]. Domain architects enjoy increased productivity when designing and analyzing systems using DSMLs tailored to the concepts, relationships, needs, and constraints of their particular domains. However, unlike "universal" modeling languages such as UML [14], which is standardized and evolved by the OMG [12], industrial DSMLs are best designed and maintained by the domain architects who employ them.

A necessary condition for the industrial adoption of DSMLs is the use of metaprogrammable tool suites that enable the reuse of complex tools for modeling, model transformation, model management, model analysis, and model integration over a wide range of domains. Today, industry managers are faced with a confounding array of tools and metamodeling languages which support the creation and use of DSMLs. The consequences of choosing one tool or metamodeling language over another are not immediately obvious. In this chapter, we examine technical solutions for metaprogrammability and analyze the relationship between metamodeling languages and metaprogrammable tool architectures. Specifically, we will provide a technical comparison of three metamodeling languages: MetaGME, Ecore, and Microsoft's Domain Model Definition language. Note that this list is by no means exhaustive; other metamodeling languages not discussed here include the Meta-Modelling Language (MML)[20] and those defined for The eXecutable Metamodeling Facility (XMF) [27].

MetaGME is the native metamodeling language supported by the Generic Modeling Environment (GME) [2], a graphical metaprogrammable modeling tool [4]. GME is developed by the Institute for Software Integrated Systems (ISIS) of Vanderbilt University. MetaGME's syntax is based on UML class diagrams [14] with OCL constraints [17].

Ecore is similar to the EMOF flavor of the pending MOF 2 specification currently being revised by OMG. It serves as the metamodeling language of the Eclipse Modeling Framework (EMF) [9]. Every Ecore metamodel may be equivalently represented through an Ecore dialect of XMI [18] or as Java source code. EMF was initially released in 2003 by IBM.

The Domain Model Definition (DMD[1]) language will be the metamodeling language supported by Microsoft's forthcoming Visual Studio Domain-Specific Language Toolkit. This toolkit realizes the Software

[1]The DMD label is not standardized by Microsoft, but we adopt here for lack of a better short-hand name for Microsoft's forthcoming metamodeling language.

Published in the Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Leung, Sang H. Son, CRC Press, 2006

Factories vision for model-based design [23]. DMD is currently still pre-release — our work in this paper is based on the September 2005 beta of the DSL toolkit.

Metamodeling languages play a highly significant role in the metaprogrammable modeling tools, which support them because they are used to create domain-specific modeling configurations of those tools. However, model-based design technology is rapidly evolving, and tool designers may need to support multiple metamodeling languages side-by-side, change the metamodeling interface exposed by a tool suite, or otherwise enable the portability of models between metamodeling languages. Recent work [22][21] has shown that model-to-model transformations can enable the portability of models across tool suites; however, understanding the impact of the selection of a metamodeling language on the reusability of such a suite of tools is also an important issue. We argue that the consequences of choosing one metamodeling language over another are fairly minor, since all of the metamodeling languages support the same fundamental concepts for DSML design. Consequently, model-to-model transformation approaches can aid in retrofitting a metaprogrammable tool suite such that it supports a new metamodeling language.

This chapter is organized as follows: **Section 2** defines metaprogramming, metaprogrammable tools, and metaprogrammable tool suites. **Section 3** compares the abstract syntax of the three metamodeling languages briefly described above to establish their fundamental similarity. We also provide some analysis of the trade-offs made when selecting one language over another in terms of the convenience afforded by each language's abstract syntax and the capabilities of the current supporting tools of each. **Section 4** outlines a set of procedures and tools which can enable the support of a new metamodeling language by an existing metaprogrammable toolsuite. The methods described depend on knowledge of the deep similarities between metamodeling languages established in Sect. 3. **Section 5** presents our conclusion.

2 Modeling Tool Architectures and Metaprogrammability

This section defines metaprogramming, metaprogrammable tools, and metaprogrammable tool suites. Traditionally, metaprogramming implies the writing of programs that write or manipulate other programs (or themselves) as their data. Metaprogramming allows developers to produce a larger amount of code and increase productivity. Although the end purpose of metaprogramming in model-based design is still devel-

Published in the Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Leung, Sang H. Son, CRC Press, 2006

oper productivity, the means is slightly different. The task of metaprogramming in model-based design is to automatically configure and customize a class of generic modeling, model management, transformation, and analysis tools for use in a domain-specific context.

The primary hurdle with 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 automobile electronic control unit) might be unjustifiably expensive. On the other hand, a general modeling tool with "universal" modeling concepts and components would lack the primary advantage offered by the domain-specific approach: dedicated, customized support for a wide variety of application domains. The solution is to use a highly-configurable modeling tool which may be easily customized to support a unique environment for any given application domain. These are known as metaprogrammable modeling tools, because users must 'program' the tool itself before they can use it to build models. The chief virtue of metaprogrammable tools is that they enable relatively quick and inexpensive development of domain-specific modeling environments (DSME) – especially when the tool customizations themselves can be easily specified through a model-based design process.

Experience with metaprogrammability indicates that metaprogrammable tools generally implement following core elements:

- A **Metaprogramming Language** that defines the core concepts and operations for customizing the generic tool into a DSME. The role of the metaprogramming language is to dictate the configuration of a generic tool to support modeling in a specific domain. Since all metaprogrammable modeling tools need to perform similar tasks (i.e., store models, allow users to access and manipulate models, and transform models into some useful product), we might expect that the metaprogramming languages associated with different tools will have a common expressive capability in terms of entity-relationship modeling. The primary differences between metaprogramming languages from different tools generally result from the model presentation capabilities of the tools themselves.

- A **Metamodeling Language** (or metalanguage) used to model (at the minimum) DSML abstract syntax, which expresses the language concepts, and relationships and well-formedness rules [17]. The primary benefit metamodeling languages provide over metaprogramming languages is that metamodel-

Published in the Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Leung, Sang H. Son, CRC Press, 2006

ing languages take a model-based approach to language specification – they operate at a higher level of abstraction from the tool. Traditionally, tools relied solely on metaprogramming languages for DSME specification. However, the community has come the realization that the interoperability of models between metaprogrammable tools suites can most easily be accomplished by agreeing on a standard metamodeling language which can abstract away the differences between various modeling tools. Consequently, the idea of decoupling the metamodeling language from the underlying metaprogramming concepts of the tool and adopting a standards-based metalanguage such as MOF [15] has come into practice. Interestingly, while all modeling tools have settled on the entity-relationship paradigm for expressing systems, there are a variety of different approaches to model presentation, broadly including:

- Full graphical - Users freely draw graphically complex entity-relationship models on a modeling palette. Model elements can be moved about the palette and can be visualized using domain-specific sprites, abstract forms, or even domain-appropriate animations.
- Restricted graphical - Users see graphical representations of the model compositional structure (generally expressed as a tree-like containment hierarchy), but the visualization itself is not domain-specific.
- Textual

The plurality of approaches to model presentation means that model presentation information is not generally portable between modeling tools. Consequently, the metalanguages closest to being true industry standards, MOF and Ecore, standardize no first-class constructs specifically modeling concrete syntax. Furthermore, XMI, the most widely-adopted model-XML serialization language, defines no elements specifically for capturing presentation metadata. Tools which support domain-specific concrete syntax as well as abstract syntax capture concrete syntax in the metaprogramming language or even in a metamodeling language (as is the case with MetaGME) [4].

- A **Mapping** that overlays the DSML abstract syntax expressed using the metalanguage onto the syntax of the metaprogramming language supported by the generic (metaprogrammable) tool. The mapping process is generally referred to as metatransformation or metainterpretation.

Published in the Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Leung, Sang H. Son, CRC Press, 2006

Formally stated, let MML be a metamodeling language, MPL be the metaprogramming language supported by the generic tool, and $DSML$ be a domain-specific modeling language. Also, let the notation ${}_{L1}A_{L2}$ denote the abstract syntax of $L2$ expressed using $L1$. The metatransformation can then be stated as follows:

$$ {}_{MML}T_{MPL} : {}_{MML}A_{DSML} \rightarrow {}_{MPL}A_{DSML} \tag{1.1} $$

Metaprogramming thus involves the following two steps:

1. Specify the DSML abstract syntax ${}_{MML}A_{DSML}$ using a metamodel

2. Apply the metatransformation ${}_{MML}T_{MPL}$ to that metamodel

The resulting abstract syntax specification of $DSML$ using MPL represents customization of the generic tool to support modeling in $DSML$.

Of course, simple metaprogrammable modeling tools as described above do not provide the full range of required capabilities for semantically-rich model-based design. These tools must be integrated with helper tools, plug-ins, and languages to provide constraint management, model transformation, model interpretation, and other capabilities [5]. A constraint manager is responsible for enforcing the well-formedness rules of DSMLs (and consequently must be metaprogrammable). Model transformers translate models of one sort into models of another sort, or generate code from models. For example, model interpreters translate domain models into 'executable' models in order to perform simulation, or into analysis input models for analysis tools. In this sense, model interpreters may be said to assign the behavioral semantics of the supported modeling language. The eXecutable Metamodelling Facility (XMF) provided by Xactium composes a set of tool-supported languages to provide rich metamodeling capabilities capable of capturing the syntax and semantics of modeling languages. This set includes an abstract syntax modeling facility which supports a MOF-like language, a concrete syntax language, a constraint language, an action language for modeling behavioral semantics, and a model transformation language [27].

In the remainder of this section, we describe the MIC suite of metaprogrammable tools: GME, UDM, GReAT, and DESERT. We elucidate the role of metaprogramming and metamodeling languages in each of

Published in the Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Leung, Sang H. Son, CRC Press, 2006

these tools.

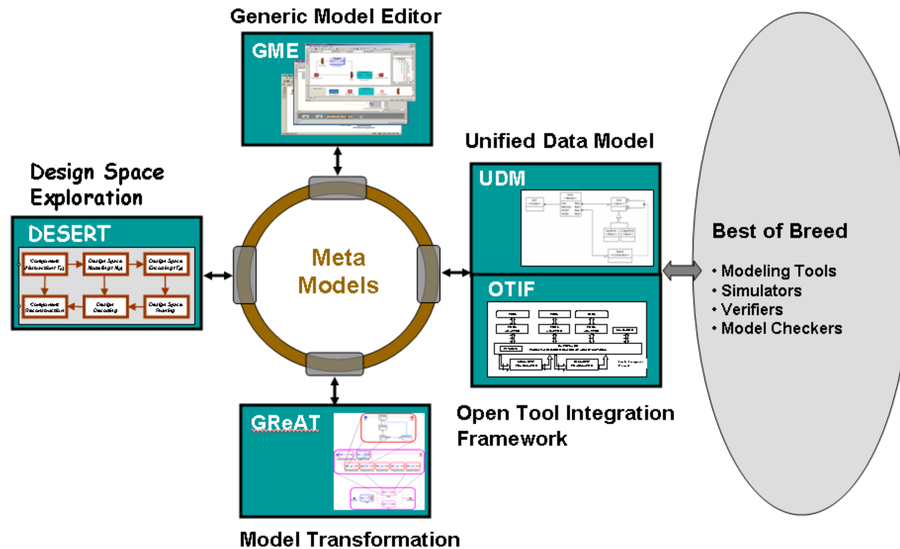

Figure 1.1: MIC Metaprogrammable Tool Suite

2.1 Metaprogrammable Modeling Tool - GME

GME is the MIC metaprogrammable graphical modeling tool [2]. It includes a graphical model builder, supports a variety of model persistence formats, and is supported by a suite of helper tools; its architecture is further depicted in Figure 1.2. The Model builder component serves as the tool front-end which allows modelers to perform standard CRUD (create, request, update, destroy) operations on model objects which are stored GME's model repository. The model builder provides access to the concepts, relationships, composition principles, and representation formalisms of the supported modeling language. The model repository is physically realized via a relational database. GME uses OCL as its constraint specification language and includes a metaprogrammable plug-in for enforcing domain constraints.

The soul of GME, however, is its metaprogramming language, which is called GMeta. The semantics of this language are implemented in the GMeta component of GME's architecture as depicted in fig. 1.2. The GMeta component provides a generic interface to the database used to implement GME's model repository. It implements the following GMeta language concepts: Folders, Models, Atoms, Sets, References, Connec-

Published in the Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Leung, Sang H. Son, CRC Press, 2006

tions, ConnectionPoints, Attributes, Roles, and Aspect[2]. All of the model manipulation operations in the GME model builder map to **domain-generic** macro manipulation operations on the objects in the GMeta component. For example, an object creation operation in the model builder maps to creation of an instance of one (or more) of the above concepts in the GMeta component of the model repository. Configuration and customization of GME via metaprogramming accomplishes the following:

- It specifies the mapping of domain modeling concepts into GMeta concepts. For example, domain relationships are specified as GMeta Connections.
- It specifies a suite of macro operations for model manipulation. For example, suppose we wrote a metaprogram which configures GME to serve as an environment for modeling FSMs. We specify that the *State* domain concept has a boolean attribute *IsInitial* which will specify whether a particular state in an FSM is the initial state. Then, creating a state instance in the model builder might result in the instantiation of a GMeta Atom and a GMeta Attribute in the GMeta component of the model repository.
- It constrains the available model operations according to domain well-formedness rules. This means that users of the language will not be able to specify domain models which defy the definitions of the domain constructs as set forth in the metaprogram.

GME currently supports MetaGME as its native metamodeling language. MetaGME is based on stereotyped UML class diagrams with OCL constraints, and serves as a layer of abstraction on top of GMeta. Users may metaprogram GME by specifying the abstract syntax of a DSML as a MetaGME stereotyped class diagram. The stereotypes in MetaGME refer to the concepts of GMeta listed above. Metatransformation maps the metamodel into these lower-level metaprogramming constructs. Formally, metatransformation in GME is a concretization of equation 1.1 as follows:

$$ {}_{MetaGME}T_{GMeta} : {}_{MetaGME}A_{DSML} \rightarrow {}_{GMeta}A_{DSML} \tag{1.2} $$

[2]W expand on these concepts in the next section.

Note that this transformation is metacircular in the sense that MetaGME can also be realized as a DSML, and GME can be metaprogrammed to provide a domain-specific modeling environment for metamodeling.

$$_{MetaGME}T_{GMeta} : {}_{MetaGME}A_{MetaGME} \rightarrow {}_{GMeta}A_{MetaGME} \tag{1.3}$$

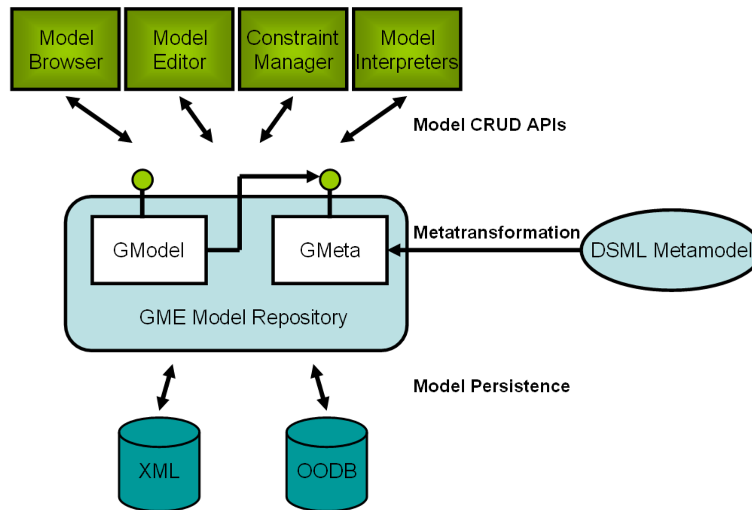

Figure 1.2: Modeling Tool Architecture: GME

2.2 Metaprogrammable Model Manipulation Management API - UDM

UDM, or Universal Data Model, is a tool framework and metaprogrammable API which can be configured to provide **domain-specific** programmatic access to an underlying network of model objects independent of the mechanism used to persist those model objects. UDM serves as a model access and manipulation layer, forming the "backbone" that model editing, database, and generator tools require [8]. It interoperates with and lends additional functionality to GME[3]. UDM uses UML class diagrams as its metamodeling language. Metaprogramming UDM thus involves specification of a DSML using UML class diagrams. Furthermore, just as GME internally represents domain models in its repository using the domain-generic GMeta constructs, UDM internally represents domain models using domain-generic UML constructs. Although UDM provides a generic UML-based API for accessing models, its chief virtue is to provide a domain-specific layer of

[3]GME actually supports several mechanisms for providing programmatic access to models, including the Builder Object network (of which there are both C++ and Java versions). For the sake of brevity, we only discuss one.

Published in the Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Leung, Sang H. Son, CRC Press, 2006

abstraction over the object models of various underlying back-end persistence technologies. UDM supports multiple such back-ends as shown in 1.3.

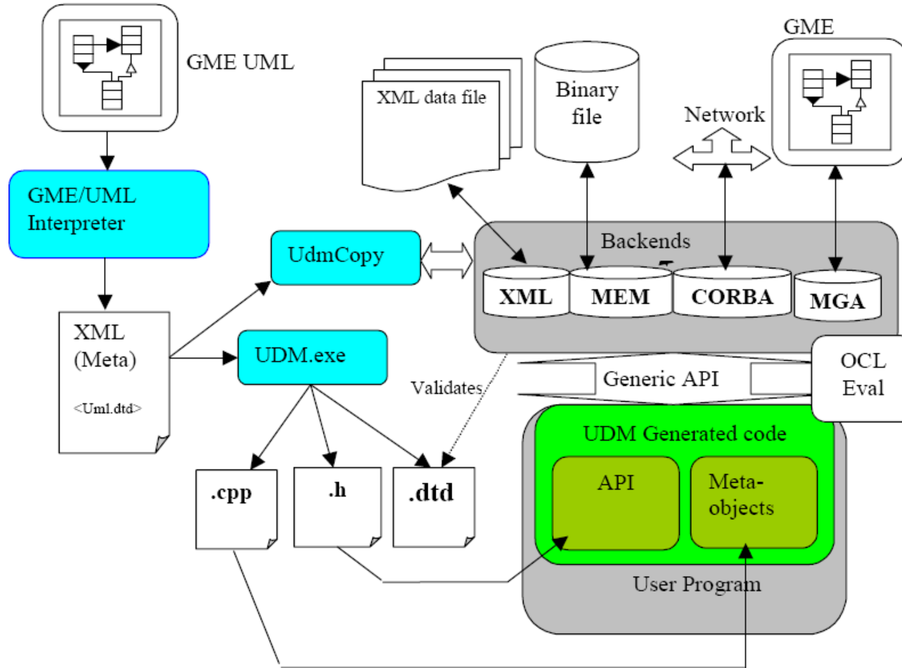

Figure 1.3: Universal Data Model Framework

With the GMeta back-end, the model object network physically resides in GME's relational internal object database. Direct access to this database is accomplished through the GMeta domain-generic API. With the XML back-end, the model object network resides in an XML file which may be manipulated through the domain-generic DOM API. The generated UDM API for a DSML is a code-level realization of the UML metamodel used to specify the abstract syntax of the DSML. This domain-specific API is a wrapper facade for the domain-generic UML API, which in turn is a wrapper facade for the domain-generic API of each supported underlying model persistence mechanism. Consequently, different metatransformations are used to metaprogram UDM for each of the supported back-ends.

For example, the metatransformation for the GMeta back-end may be formalized as:

$$_{UML}T_{GMeta} : {}_{UML}A_{DSML} \rightarrow {}_{GMeta}A_{DSML} \tag{1.4}$$

It is often useful to construct a domain-specific UDM API which supports a DSML originally specified

using MetaGME rather than UDM's implementation of UML. Consequently, the UDM set of tools includes the MetaGME2UML model-to-model transformation:

$$_{MetaGME}T_{UML} : _{MetaGME}A_{DSML} \rightarrow _{UML}A_{DSML} \tag{1.5}$$

Clearly, UDM was designed with an eye toward the manipulation of models irrespective of the platform on which they were developed or the mechanism used to persist them. Because of its relationship with UML, UDM even supports XMI [18] import and export of models and metamodels.

2.3 Metaprogrammable Design Space Exploration - DESERT

When large-scale systems are constructed, in the early design phases it is often unclear what implementation choices could be used in order to achieve the required performance. In embedded systems, frequently multiple implementations are available for components (e.g. software on a general purpose processor, software on a DSP, FPGA, or an ASIC), and it is not obvious how to make a choice, if the number of components is large. Another metaprogrammable MIC tool can assist in this process. This tool is called DESERT (for Design Space Exploration Tool). DESERT expects that the DSML used allows the expression of alternatives for components (the design space) in a complex model.

Once a design space is modeled, one can attach applicability conditions to the design alternatives. These conditions are symbolic logical expressions that describe when a particular alternative is to be chosen. Conditions could also link alternatives in different components via implication. One example for this feature is: "if alternative A is chosen in component C1, then alternative X must be chosen in component C2". During the design process, engineers want to evaluate alternative designs, which are constrained by high-level design parameters like latency, jitter, power consumption, etc. Note that these parameters can also be expressed as symbolic logic expressions. DESERT provides an environment in which the design space can be pruned and alternatives rapidly generated and evaluated.

DESERT consumes two types of models: component models (which are abstract models of simple components) and design space models (which contain alternatives). Note that for design space exploration the internals of simple components are not interesting, only a "skeleton" of these components is needed. The

Published in the Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Leung, Sang H. Son, CRC Press, 2006

design space models reference these skeletons. DESERT uses a symbolic encoding technique to represent the design space that is based on Ordered Binary Decision Diagrams (OBDD-s) [6]. OBDD-s are also used to represent applicability constraints, as well as design parameters. The conversion to OBDD-s happens in an encoding module.

Once the symbolic representation is constructed, the designer can select which design parameters to apply to the design space and thus "prune away" unsuitable choices and alternatives. This pruning is controlled by the designer, but it is done on the symbolic representation: the OBDD structures. Once the pruning is finished, the designer is left with 0, 1, or more than one designs. 0 means that no combination of choices could satisfy the design parameters, 1 means a single solution is available, and more than one means multiple alternatives are available that cannot be further pruned based on the available information. This latter case often means that other methods must be used to evaluate the alternatives, e.g. simulation. The result of the pruning is in symbolic form, but it can be easily decoded and one (or more) appropriate (hierarchical) model structure reconstructed from the result.

DESERT is metaprogrammable as it can be configured to work with various DSML-s (however all of them should be capable of representing alternatives and constraints). The metaprogramming here happens in two stages: one for the model skeleton generation, and the other one for the model reconstruction. DESERT has been used to implement domain-specific design space exploration tools for embedded control systems and embedded signal processing systems.

2.4 Metaprogrammable Model Transformations - GReAT

The Graph Rewriting And Transformation language (GReAT) is the MIC model-to-model transformation language[1][11]. 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

Published in the Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Leung, Sang H. Son, CRC Press, 2006

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.

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.

Ultimately, GReAT transformation models are used to generate C++ code which uses automatically-generated domain-specific UDM APIs for the source and destination metamodels to programmatically execute model-to-model transformations. Consequently, GReAT inherits its metaprogrammability from UDM. The source and destination metamodels accepted by GReAT are expressed using UMD-style UML class diagrams.

Formally stated, let SRC and DST be respectably the source and destination metamodels of GReAT Transformation ${}_{SRC}T_{DST}$, and let IN and OUT be arbitrary input and output domain models expressed

using the languages defined by SRC and DST. Then, assuming that the GMeta back-end is always used for persistent model storage, any execution of ${}_{SRC}T_{DST}$ may be formalized as the following series of transformations:

$$
\begin{aligned}
{}_{IN}T_{OUT} : ((((&{}_{GMeta}A_{IN} \rightarrow {}_{UML}A_{IN}) \rightarrow {}_{SRC}A_{IN}) \\
&\rightarrow {}_{DST}A_{OUT}) \rightarrow {}_{UML}A_{OUT}) \rightarrow {}_{GMeta}A_{OUT}
\end{aligned}
\tag{1.6}
$$

To summarize, the MIC metaprogrammable tool suite allows users to:

- Specify a domain-specific modeling language, include a concrete and abstract syntax and domain constraints using GME and MetaGME
- Build system models using the metamodeled DSML and verify that the models do not violate the domain constraints
- Construct model interpreters to parse and analyze the system models using UDM
- Transform the system models of into models expressed using a different DSML using GReAT; a specialized case of this is generating code directly from models to begin implementing the system
- Formally express the semantics of the DSML by mapping it onto another modeling language that expresses a formal model of comutation using GReAT
- Perform design-space exploration on the modeled system using DESERT.
- Serialize the models to or import models from XMI using UDM

The MIC toolsuite supports full-featured, semantically rich model-based design. Furthermore, we note that it is a framework which already utilizes multiple metamodeling languages mediated by a model-to-model transformation: MetaGME and UDM UML mediated by the MetaGME2UML transformation.

Published in the Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Leung, Sang H. Son, CRC Press, 2006

3 A Comparison of Metamodeling Languages

This section provides a brief overview of three metamodeling languages: MetaGME, Ecore, and Microsoft's Domain Model Designer (DMD) language. We compare these languages to draw out the close kinships and similar expressive power they share via their common application of UML-like Object-Oriented principles to abstract syntax modeling. We phrase their differences in terms of trade-offs for selecting one language over another.

We discuss how each of these languages captures in some way a common core set of metamodeling constructs derived from UML[14]:

- **Classes** are types whose instances have identity, state, and an interface. They are used to specify the concepts of a modeling language. The state of a class is expressed by its attributes and its interface is defined by the associations in which it participates. Class definitions may be extended through the use of inheritance. Some languages support multiple inheritance; some support only single inheritance.
- **Associations** describe relationships between classes, including composition. Associations may be binary or n-ary, and may include attributes or not, depending on the language.
- **Data Types** are non-instantiable types with no object identity, such as primitive types and enumerations, which are used to type attributes. All three metamodeling languages we consider support a common core set of primitive data types, and some also support additional data types. The common set includes strings, integers, doubles, booleans, and enumerations.

3.1 MetaGME

MetaGME is the metamodeling language for the Generic Modeling Environment[4], which was discussed at length in Section 2. MetaGME is a graphical language, and its concrete syntax is derived from UML class diagrams augmented with UML class stereotypes. Figure 1.4 is a UML class diagram which captures a simplified metamodel for MetaGME.

FCO and its subtypes capture the class concept (multiple inheritance is allowed). Connections strongly resemble UML association classes – they may define binary stateful connections. MetaGME provides only the

Published in the Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Leung, Sang H. Son, CRC Press, 2006

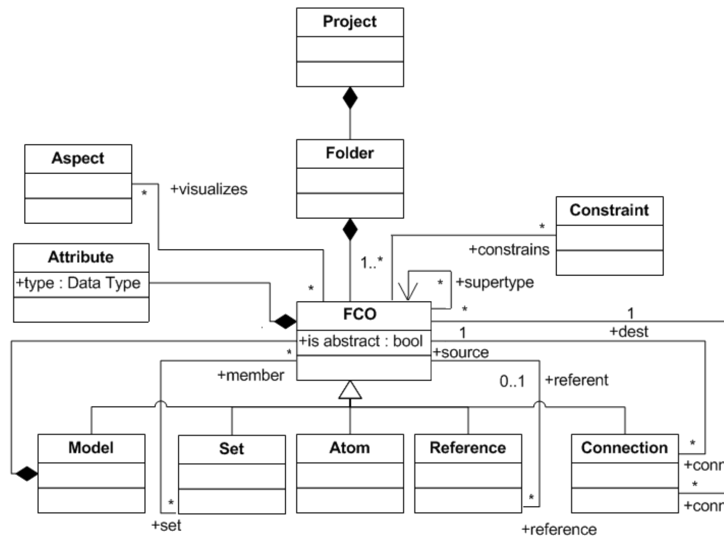

Figure 1.4: Simplified Version of the MetaGME Metamodel

common core set of data types (strings, integers, doubles, booleans, and enumerations) for typing attributes in metamodels. Users can capture domain constraints directly in a MetaGME metamodel using OCL, and these constraints will be parsed, interpreter, and enforced by the GME Constraint Manager plug-in.

MetaGME allows users to model the concrete syntax, abstract syntax, and syntactic mapping of a DSML all in the same metamodel. Users can tailor the appearance of each association using the attributes of the Connection class, and they can specify icons for language concepts using the attributes of FCO and its descendants. Furthermore, the MetaGME class stereotypes allow users to specify another element of DSML concrete syntax – the concepts represented by the stereotypes have been refined to support the inclusion of commonly-used graphical modeling idioms or design patterns in a DSML, which are automatically supported by GME. These stereotypes are defined as follows[10]:

- *FCOs* (First-Class Objects) are classes which must be abstract but can serve as the base type of a class of any other stereotype in an relationship. For example, Atoms may only inherit from other Atoms and FCOs, but a particular FCO might serve as the base type of both Atoms and Models.
- *Models* are composite classes. An attribute on the Containment connection allows users to specify

Published in the Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Leung, Sang H. Son, CRC Press, 2006

whether a given component type is graphically exposed by its containing Model type as a *port*.

- *Atoms* are elementary, non-composite classes.
- *Sets* are classes which group other classes together using a special meta-level relationship which captures the semantics of UML non-composite aggregation.
- *References* are classes that refer (through a special meta-level relationship) to other classes.
- *Connections* are analogous to UML association classes.
- *Aspects* are classes which group other classes together (through a special meta-level relationship) into logical visibility partitions to present different views of a model.

GME provides special default visualizations for languages that use various patterns of these stereotypes. For example, the default visualization for the Modular Interconnect pattern appears in Figure 1.5. The metamodel fragment use to specify this visualization appears in Figure 1.6, where the Port class is composed into the Module class as a port. This visualization could be further customized by specifying icons for Module and Port and tailoring the appearance of PortConnection in the metamodel.

Finally, MetaGME provides robust support for metamodel composition. The GME tool itself allows the importation of one model into into another, and MetaGME includes a special set of "class equivalence" operators for specifying the join-points of the composed metamodels. It also defines two additional inheritance operators to aid in class reuse: implementation inheritance and interface inheritance[3]. Implementation inheritance allows a derived class to inherit only the attributes and component classes of its base type, while interface inheritance allows a derived class to inherit only the non-composition associations of its base type.

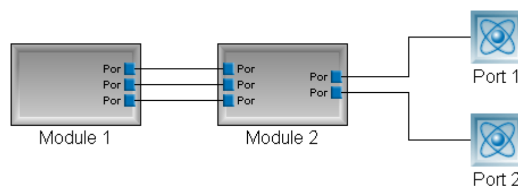

Figure 1.5: Modular Interconnected in GME

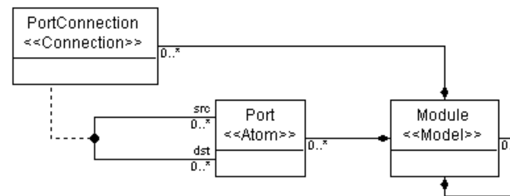

Figure 1.6: Metamodel Fragment Specifying Modular Interconnection

3.2 Ecore

Ecore is the metamodeling language for the Eclipse Modeling Framework[9]. The domain-specific modeling environments built using EMF run as plug-ins for the Eclipse platform. The EMF toolsuite includes the following tools:

- The Ecore language for specifying the abstract syntax of domain-specific modeling languages. The framework supports import and export with an Ecore-flavored dialect of XMI (it can also import standard MOF XMI).
- EMF.Edit, which includes a domain-generic set of classes used to construct domain-specific modeling environments
- EMF.Codegen, the metatranslator which renders a DSME and a domain-specific model CRUD API from an Ecore metamodel
- A domain-generic API for accessing and manipulating EMF models

The DSMEs produced by EMF "out of the box" all share a domain-generic hierarchy-based tree visualization, and EMF does not provide tool support for modeling concrete syntax. Complicated domain-specific visualizations can be created by extending the generated DSME or by hand-coding an editor using the Eclipse Graphical Editing Framework (GEF). EMF lacks native tool support for modeling domain constraints, model migration, and multi-aspect (or multi-view) modeling. It is possible to perform metamodel composition, however, by importing one metamodel into another. Although it is not possible to capture domain constraints in an Ecore metamodel, it is possible to hand-weave constraints into the Java code generated from a metamodel using a Java-based OCL library developed at Kent University[25].

The Ecore metamodeling language itself strongly resembles the EMOF flavor of the MOF 2.0 specification [16]. However, EMF takes a more bottom-up, code-centric, and Java-specific view of modeling than is reflected in the MOF standard, which results in a few simplifications in Ecore. Ecore does not support class nesting or a first-class association construct, but it does support multiple inheritance[4]. A simplified UML class diagram for the Ecore metamodel (as it relates to domain-specific modeling) is provided in figure 1.7[5].

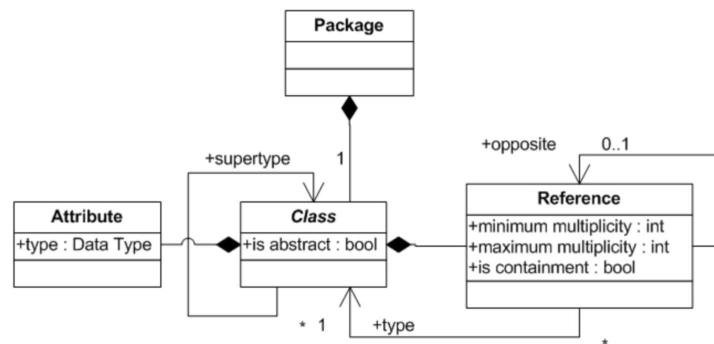

Figure 1.7: Simplified Version of the Ecore Metamodel

Because Ecore lacks a first-class association construct, association is expressed using references. Each class forming one end of a logical association may have a reference to the class which forms the other end of the logical association. References may be used to represent both symmetric and asymmetric association: in the symmetric case, each end class possesses a reference to the other end class. In asymmetric association, only one of the end classes possesses a reference to the other end class. Ecore's representation of associations is sufficiently expressive to capture the information relevant to DSML design: class relation, association roles, composition, and navigability. Clearly, however, Ecore cannot represent associations with state. This certainly makes many modeling languages more tedious to define, but does not decrease the relative expressive power of the Ecore. Every association with attributes can be modeled as two associations which link to a Class with the same attributes.

[4]When an Ecore class with multiple base types is serialized to Java, the corresponding class extends one of those base classes and implements the others as mix-in interfaces.

[5]The Ecore metamodeling constructs are all more properly named with 'E' preceding the construct title (for example, "EClass" instead of "Class"). We omit these preceding 'E'-s.

Through its close relationship with Java, Ecore enables a vast array of data types. In addition to all of the usual primitive types, Ecore allows bytes, shorts, and in general any Java class to be wrapped as a data type. Many of these additional types, such as collections, structures, and arbitrary classes, are of limited use in the context of graphical DSMLs because of the difficulty of providing a simple graphical interface for setting the values of attributes with complex types. For the most part, these only come in to play when using Ecore itself as a DSML for defining logical data models or database schemas.

3.3 Microsoft Domain Model Designer

Microsoft DMD is part of the Microsoft Domain-Specific Languages toolsuite[26]. This new toolsuite will realize the Software Factories vision of a workbench for creating, editing, visualizing, and using domain-specific data for automating the Windows-based enterprise software development process. The domain-specific modeling environments produced by MS DSL plug in to Visual Studio. For the sake of clarity, we should point out that with its new tool, Microsoft has introduced new terminology for talking about familiar domain-specific modeling concepts:

- **Domain Model:** a metamodel
- **Designer:** a domain-specific modeling environment
- **Designer Definition:** an XML-based metaprogramming specification for the DSL tool suite which includes concrete syntax and syntactic mapping information

Currently, the metaprogrammable DSL toolsuite consists of the following major components:

- The Domain Model Designer metalanguage/tool, which is used to formulate DSL abstract syntax
- The Designer Definition language, which is used to specify the DSL concrete syntax, and some primitive language constraints
- A metatranslator, which generates from a domain model and a designer definition a designer and a domain-specific API for model access and manipulation
- The Meta Data Facility (MDF) API, which provides generic CRUD access to in-memory model storage

Published in the Handbook of Real-Time and Embedded Systems, Ed. Insup Lee, Joseph Leung, Sang H. Son, CRC Press, 2006

- A tool for creating template-based model parsers with embedded C# scripts which may be used for code generation

Because of the current immaturity of this framework, many supporting tools remain unimplemented. Microsoft eventually plans to provide tool support DSL constraint specification, model migration, language composition, and multi-aspect (or multi-view) modeling.

Although Microsoft provides no explicit meta-metamodel, it is easy to derive one from experimentation with their tools. We provide a simplified visualization of this meta-metamodel using a UML class diagram in figure 1.8.

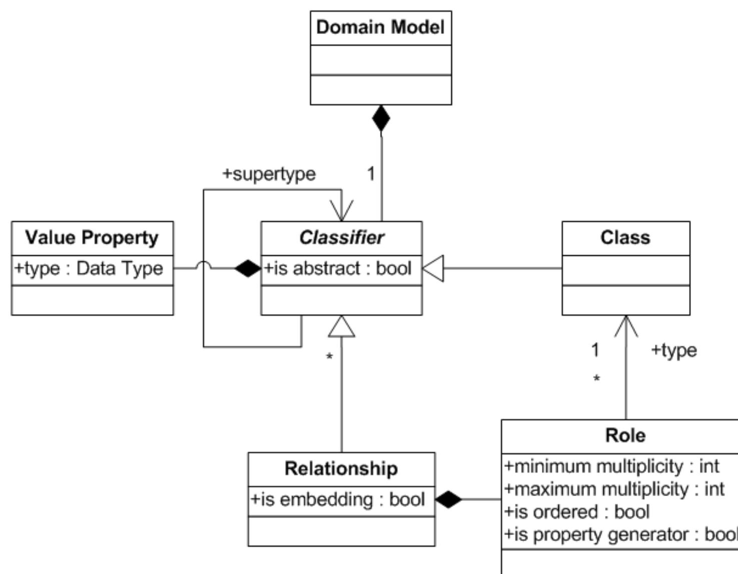

Figure 1.8: Simplified Version of the Microsoft Domain Model Designer Metamodel

Notice that the language does not allow multiple inheritance, but does include the notion of stateful associations (Relationships). These Relationships come in two varieties; Reference Relationships merely denote association, while Embedding Relationships denote composition. DMD also captures a variety of data types, including the standard set of primitive datatypes, longs, DateTimes, and GUIDs. As stated above, the Microsoft DSL tools currently do not provide a modeling interface for capture concrete syntax – that is done in XML through a designer definition.