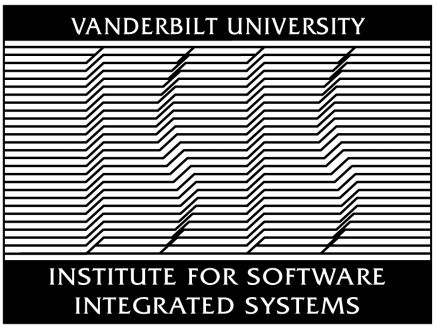


Institute

for

Software-Integrated Systems

Technical Report

TR #: **ISIS-00-200**

Title: **Formalizing the Specification of Graphical Modeling Languages**

Authors: **Greg Nordstrom, Akos Ledeczi**

Abstract

Model integrated computing (MIC) is an effective and efficient method for developing, maintaining, and evolving large-scale computer-based systems (CBSs). Systems are synthesized from models created using customized, domain-specific model integrated program synthesis (MIPS) environments. The MultiGraph Architecture (MGA), developed by Vanderbilt University's Institute for Software Integrated Systems, is a toolset for creating such graphical MIPS environments.

Until now, these MIPS environments have been hand-crafted specifically for each domain. Because common modeling concepts appear in many, if not all, MGA-based MIPS systems, research suggests it is possible to "model the modeling environment" by creating a *metamodel* – a model that formally describes a particular domain-specific MIPS environment (DSME). By modeling the syntactic, semantic, and presentation requirements of a DSME, the metamodel can be used to synthesize the DSME itself, enabling design environment evolution in the face of changing domain requirements. Because both the domain-specific applications and the DSME are designed to evolve, efficient and safe large-scale computer-based systems development is possible over the entire lifetime of the CBS.

This report presents a method to represent DSME requirements using UML class diagrams and predicate logic constraint language expressions, discusses automatic transformation of metamodel specifications into DSMEs, and includes an example.

KEYWORDS

Modeling, Metamodeling, Graphical Modeling Languages, UML, OCL, MultiGraph Architecture, Modeling Environments, Model-Integrated Computing.

ACKNOWLEDGMENTS

This work was sponsored by the Defense Advanced Research Projects Agency, Information Technology Office, as part of the Evolutionary Design of Complex Software program, under contract #F30602-96-2-0227.

I. Introduction

Large computer-based systems (CBSs), where functional, performance, and reliability requirements demand the tight integration of physical processes and information processing, are among the most significant technological developments of the past 20 years [1]. CBSs operate in ever-changing environments, and throughout the system's life cycle, changes in mission requirements, personnel, hardware, support systems, etc., all drive changes to the CBS. Rapid reconfiguration via software has long been seen as a potential means to effect rapid change in such systems. Examples of such environments include large-scale production facility process monitoring; real-time diagnostics and analysis of manufacturing execution systems; web-based information distribution and management; surety of high consequence, high reliability systems; and fault detection, isolation, and recovery of space vehicle life support systems.

Due to the complex nature of large-scale, mission-critical systems, software modification involves a large amount of risk. The magnitude of this risk is proportional to the size and complexity of the system, not to the size of the change. Small modifications in one area can cause large and unforeseen changes in others. Because such risk is always present, it must be managed. To effectively manage such risk, the entire system must be *designed* to evolve. Key factors in this evolution are:

- **Requirements capture**: A method to state the CBS's requirements and design in concise, unambiguous terms.
- **Program synthesis**: The ability to automatically transform requirements and design information into application software.
- **Application evolution**: A method to safely and efficiently evolve the application software over time as system requirements change.
- **Design environment evolution**: A method to ensure the design environment (e.g. design and analysis tools, etc.) can correctly model domain-specific systems as domain requirements change.

An emerging technology that enables such evolution is model integrated computing (MIC). MIC allows designers to create *models* of domain-specific systems, validate these models, and perform various computational transformations on the models, yielding executable code, configuration data, or input data streams for simulation and/or analysis tools.

One approach to MIC is model-integrated program synthesis (MIPS). In MIPS, formalized models are created that capture various aspects of a domain-specific system's desired structure and behavior. Model *interpreters* are used to perform the transformations necessary to synthesize executable code for use in the system's execution environment, often in conjunction with code libraries and some form of middleware (e.g. CORBA [18], the MultiGraph kernel [2], POSIX [19], etc.), or to supply configuration information or input data streams to various GOTS, COTS, or custom software packages (e.g. spreadsheets, simulation engines, etc.) When changes in the overall system require new application programs, the models are updated to reflect these changes, the interpretation process is repeated, and the applications and data streams are regenerated automatically from the models.

The MultiGraph Architecture (MGA) is a toolset for creating domain-specific MIPS environments. Although the MGA provides a means for evolving domain-specific applications, such capability is generally not enough to keep pace with large changes in systems requirements. Throughout the

lifetime of a system, particularly a large-scale system, requirements often change in ways that force the entire design environment to change. For example, if a domain-specific MIPS environment (DSME) exists for modeling a chemical plant and is used to generate executable code for use on the plant's monitoring and analysis computers, what happens when new equipment is later added to the plant – equipment that was not in use or was unheard of at the time the DSME was created? Generally, the existing DSME would not be able to model new configurations of the plant, and the entire DSME would have to be upgraded to allow models containing the new equipment to be incorporated into existing and future plant designs.

Until now, DSMEs were handcrafted, and rebuilding a DSME was a long and costly process. Our approach is to *automatically generate the DSME* by applying MIPS techniques to the process of creating the DSME itself – to "model the modeling environment" in a manner similar to modeling a particular domain-specific application. (In fact, a DSME *is* a domain-specific application, where the domain is the set of all possible MIPS environments). Just as domain-specific models are used to generate domain-specific applications, by adding a *metaprogramming interface* to a MIPS environment, the MIPS environment can be used to generate various DSMEs. Such a MIPS environment is called a *metamodeling environment*. Because models created using a metamodeling environment describe other modeling systems, they are called *metamodels*. Metamodels are formalized descriptions of the objects, relationships, and behavior required in a particular DSME. It can be seen that this approach to DSME design and evolution is similar to that of evolving domain-specific applications using DSMEs – just "up one level" in the design hierarchy.

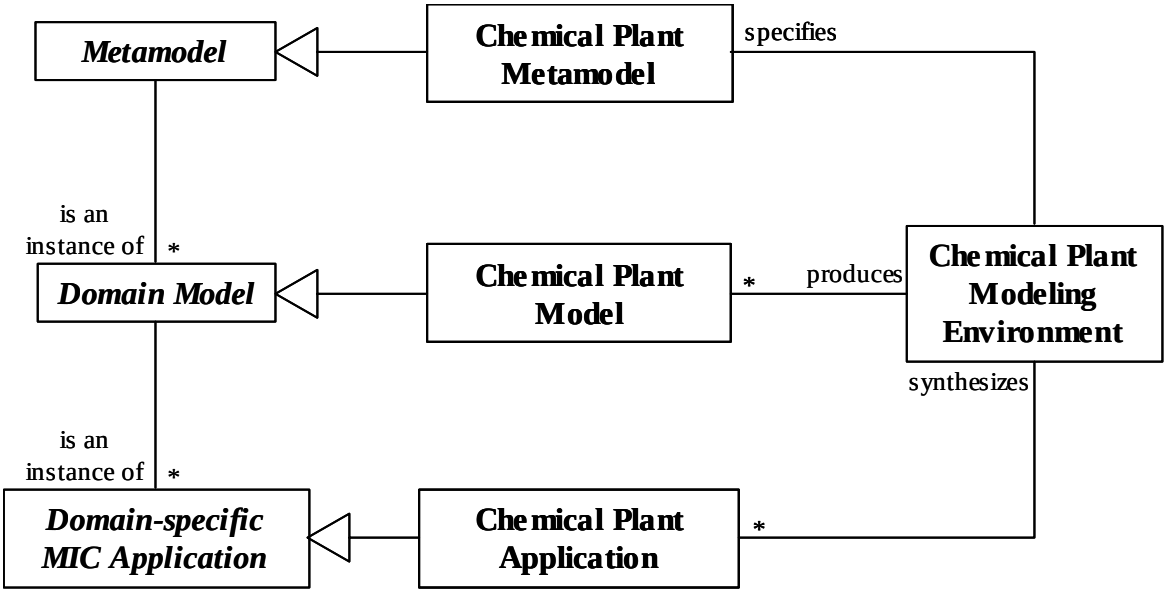


Figure 1: Modeling and metamodeling relationships

Figure 1 illustrates the relationships between the conceptual notions of metamodels, domain models, and domain-specific MIC applications and the more concrete components of an actual MIC application – in this case a chemical plant modeling environment. On the left, domain models are shown to be instances of metamodels, and domain-specific MIC applications are shown to be

instances of domain models. Said another way, a metamodel is used to specify all possible domain models, and a domain model is used to specify all possible domain-specific applications.

The center of Figure 1 shows specialized versions of the metamodel, model, and domain-specific MIC application objects used in chemical plant modeling. A chemical plant metamodel specifies the chemical plant modeling environment (shown on the right). The chemical plant modeling environment is used to produce chemical plant models and to synthesize chemical plant applications from those models. Note that the general relationships between metamodels, domain models, and domain-specific MIC applications still hold – the chemical plant model is one instance of all possible chemical plants specified by the chemical plant metamodel, and the chemical plant application is one instance of all possible chemical plant model applications.

This paper presents a method for creating metamodels to represent DSME requirements using UML object diagrams and predicate logic constraint language expressions, discusses automatic transformation of such metamodels into DSMEs, and presents an example illustrating key phases of this process.

II. Model-integrated program synthesis

Modeling reduces the design cycle time, allows completeness and consistency checking throughout the design process, aids in documenting the design, and, in the case of executable models, allows automated design validation and/or simulation. Because modeling lowers cost and error rates, it becomes a key strategy in any system design process [3]. The artifacts of the modeling process are *models* – abstractions of the original system. A key feature of a model is its ability to reduce or hide complexity. To aid designers in creating models of hardware and software systems, various modeling languages and design environments have been created. For such languages to be successful, they must be specific enough to enable designers to represent the key elements of various designs without undue constraint, while remaining general enough to allow a fairly wide variety of models to be created.

Modeling languages exist for many domains, serving many purposes. Some modeling languages are more suited to the task of hardware and software modeling than others. This paper concentrates on languages designed to model CBSs. The functional, performance, and reliability requirements of CBSs demand tight integration of physical systems with information systems [4]. When modeling such systems, it is important that these requirements be captured in a *unified* set of models. In this way each aspect of the system's overall behavior (e.g. the functional, performance, reliability, etc.) can be examined as a "separate but integrated" part of the overall CBS. There are several criteria for choosing a modeling language. These include:

- the ability to model general engineering entities, attributes, and relationships,
- acceptance of the language as a standard,
- support for a variety of modeling techniques (e.g. StateCharts, Petri-nets, etc.),
- model composition and reuse capability,
- modeling language extensibility,
- support for an automated design environment,
- the ability to conduct formal analysis and simulation of models,
- the ability to assign semantic meaning to models,
- transformation of structural and behavioral models into executable models, and
- metalanguage (formalized description of the modeling language) support.

A MIPS environment operates according to a domain-specific *modeling paradigm* – a set of requirements that governs how *any* system in the particular domain is to be modeled. These modeling requirements specify the types of entities and relationships that can be modeled; how to model them; entity and/or relationship attributes; the number and types of views or aspects necessary to logically and efficiently partition the design space; how semantic information to is to be represented in, and later extracted from, the models; any analysis requirements; and, in the case of executable models, run-time requirements.

Once a modeling paradigm has been established, the MIPS environment itself can be built. A MIPS environment consists of three main components: (1) a domain-aware *model builder* used to create and modify models of domain-specific systems, (2) the *models* themselves, and (3) one or more *model interpreters* used to extract and translate semantic knowledge from the models.

III. Metamodeling Concepts

More and more, the prefix *meta* is being attached to words that describe various modeling and data representation activities (e.g. metaprocess, metadata, metarelation, , etc.) Unfortunately, the term "meta" is not always applied consistently, causing considerable confusion among researchers. Therefore, in the context of this paper, the following definitions apply:

- **Model**: An abstract representation of a CBS.
- **Modeling Environment**: A system, based on a modeling paradigm, for creating, analyzing, and translating domain-specific models.
- **Metamodel**: A model that formally defines the syntax, semantics, presentation, and translation specifications of a particular domain-specific modeling environment.
- **Metamodeling Environment**: A tool-based framework for creating, validating, and translating metamodels.
- **Meta-metamodel**: A model that formally defines the syntax, semantics, presentation, and translation specifications of a metamodeling environment.

In a very real sense, modeling and metamodeling are identical activities – the difference being one of interpretation. Models are abstract representations of real-world systems, and when the system being modeled is *the system for creating other models*, the modeling activity is correctly termed metamodeling. Therefore, concepts that apply to modeling also apply to metamodeling. This logic can be extended to the process of meta-metamodeling, too. However, because of the goals of modeling, metamodeling, and meta-metamodeling are quite different, a four-layer conceptual framework for metamodeling has been established [5] [6] [7] and is in general use by the metamodeling community. The following table, taken from [7], describes each layer:

Table 1: Four-layer metamodeling architecture

Layer	Description
Meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for describing metamodels.
Metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.
Model	An instance of a metamodel. Defines a language to describe an information domain.
User objects	An instance of a model. Defines a specific information domain.

This four-layer architecture creates an infrastructure for defining modeling, metamodeling, and meta-metamodeling languages and activities, and provides a basis for future metamodeling language extensions. The architecture also provides a framework for exchanging metamodels among different metamodeling environments – critical for tool interoperability, since such interoperability depends on a precise specification of the structure of the language [7]. The previous definitions for Meta-metamodel, Metamodel, and Model correspond to the upper three layers of Table 1.

A. Modeling Syntax, Semantics, and Presentation

To properly capture the *syntax* of a modeling language, a metamodel must describe all entities, relationships, and attributes that may exist in the target language. As discussed in [7], when specifying graphical modeling languages, an *abstract syntax* – a language syntax devoid of implementation details – is first specified. Then a *concrete syntax* is defined as a mapping of the graphical notation onto the abstract syntax, clearly defining the particular graphical idioms and constructs used to represent entities, relationships, and attributes defined in the abstract syntax. Furthermore, in cases where graphical models are to be viewed from different *aspects* (i.e. points of view), the metamodel must clearly define a partitioning of the graphical constructs into these aspects. Such a language is called a *multi-aspect* graphical modeling language.

In addition to specifying the syntax of a modeling language, language *semantics* must also be specified in a metamodel. It is necessary to distinguish among two types of semantics – *static* and *dynamic*. Static semantics refer to the well-formedness of constructs in the modeled language, and are specified as invariant conditions that must hold for any model created using the modeling language. Dynamic semantics, however, refer to the interpretation of a given set of modeling constructs at run-time. Only static semantics may be specified in a metamodel – the metamodel has no way of knowing what meaning to associate with particular model instances after they begin to interact with an execution environment.

Distinguishing between static and dynamic semantics is best illustrated by an example. Consider a MIPS environment used to model real-time scheduling systems. A metamodel description of such a MIPS environment would necessarily define objects and relationships such as *tasks*, *events*, and *schedules*. The static semantics expressed in the metamodel would include constraints that must

be maintained to ensure a given model is valid. Scheduling system models that violate these constraints are, by definition, invalid. One such constraint, stated using standard English, might be:

- "Task duration must be specified"

Such a constraint represents part of the static semantics of the modeling language, and can be defined in the metamodel (possibly by defining a `duration` attribute associated with a `task` object and ensuring that `duration` has an appropriate value associated with it). However, the following constraint defines a dynamic semantics, and as such, cannot be represented as a metamodel constraint:

- "All tasks must meet their execution deadlines"

Here, the metamodel cannot specify that all tasks must meet their execution deadlines, since schedulability is a function of certain run-time information that is not yet known when the metamodel is created.

Another consideration in any metamodeling language is how such invariant constraint statements are specified. Constraints should analyzable, allowing automated or semi-automated consistency checking before a metamodel is used to synthesize a modeling environment. This requires that the constraints be precisely stated using a mathematical language, such as predicate calculus, where invariants take the form of Boolean expressions – expressions that must be satisfied by any instance model created using the DSME.

After the syntax and semantics of a domain-specific modeling language have been specified, a presentation specification is created that associates the syntactic and semantic specifications previously specified to a particular target modeling environment. Part of this specification is the mapping of graphical modeling idioms available in the target environment onto the abstract syntax discussed earlier. Another part of the presentation specification involves deciding how best to represent the syntactic and semantic specifications graphically. For example, if a target modeling environment supports part-whole hierarchy through the use of object containment, one may use this modeling environment feature as a mechanism for representing aggregation. Of course, other choices may exist, such as representing containment as a special type of interconnection. The choice rests with the metamodeler, given the capabilities of a particular graphical modeling environment.

As stated earlier, making modeling (and metamodeling) tools interoperable requires that metamodels be exchanged among various metamodeling tool suites. This requires that the structure of any language – its syntax and semantics – be precisely specified in the metamodel, apart from the presentation specification. In this respect, the presentation specification becomes an implementation detail that depends on the particular editing environment that is being mapped onto the syntactic and semantic specifications. Therefore, metamodeling tools and environments must be able to accept metamodels specified in a variety of metalanguages – or such metalanguages must be translatable to a metalanguage that the metamodeling environment understands. This again underscores the need to separate the implementation details from the language specification itself.

B. Model Composition, Validation, and Translation

Because common modeling concepts apply to a wide variety of engineering domains, the approach to creating DSMEs is to customize (i.e. configure) a general graphical modeling

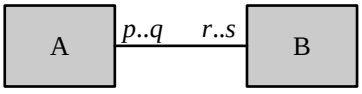
environment for use in a particular domain according to specifications included in a metamodel. This is done by representing general modeling principles abstractly and placing such representations in a repository or library. The metamodeler then accesses these representations and composes a metamodel as dictated by the modeling paradigm. Such an approach allows quick and accurate construction of metamodels – assuming, of course, that these individual representations have been validated *a priori*, and that the act of combining or composing them does not negate their individual validations (or that re-validation can be easily accomplished).

Table 2: General modeling techniques

Name	Description
Module Interconnect	Provides rules for connecting objects together and defining interfaces. Used to describe relationships among objects[8].
Aspects	Enables multiple views of a model. Used to allow models to be constructed and viewed from different "viewpoints."
Hierarchy	Describes the allowed encapsulation and hierarchical behavior of model objects. Used to represent information hiding.
Object Association	Binary and n-ary associations among modeling objects. Used to constrain the types and multiplicity of connections between objects.
Specialization	Describes inheritance rules. Used to indicate object refinement.

Table 2 describes several general modeling techniques. These techniques represents *constraints* on the modeling process. Because of their general nature, the techniques must be customized before being used in a given metamodel. This allows the metamodeler to inject domain-specific concepts into the metamodel. Said another way, this allows domain-specific constraints to be applied to specific types of models. One approach to this customization is parameterization.

Figure 2 defines a general constraint on binary object association (shown both graphically and textually). This constraint specifies that an object of type *A* can be associated with between *r* and *s* objects, inclusive, of type *B*, and that objects of type *B* can be associated with between *p* and *q* objects, inclusive, of type *A*.




```
ObjAssociation(A, p, q, B, r, s) {
  // f is a parameterized, invariant
  // Boolean expression representing a
  // general object association constraint.
  return f(A,p,q,B,r,s);
}
```

Figure 2: General object association constraint (shown graphically and textually)

Now consider an aircraft in-flight safety system modeling environment, where between three and six engine temperature sensors can be associated with a single fire suppression system actuator. The general object association constraint in Figure 2 is parameterized for this particular domain as follows:

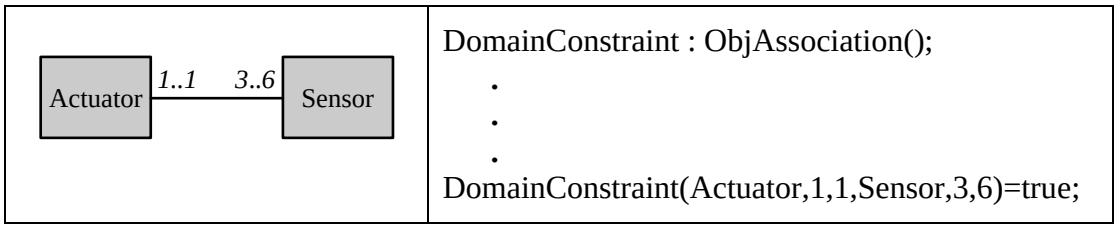


Figure 3: Customizing the general object association constraint

Figure 3 shows an instance of the general object association constraint, parameterized for the specific domain. In this case, objects A and B become Actuator and Sensor, respectively, and specific values are assigned to the variables *p*, *q*, *r*, and *s*.

Tailoring general modeling concept representations in this manner represents a specification of the syntax of the modeling language. However, there must also be a mechanism for specifying the semantics of the language. This is done by directly including additional domain-specific constraints that, even in their general form, pertain only to the domain being modeled. Such constraints would not be present in any general modeling constraints library, as they only apply to the particular domain.

To make metamodel construction efficient, the metamodeler should not have to "start from scratch" when specifying the various syntactic- and semantic constraints that form a metamodel. Rather, the metamodeler should be able to *compose* the metamodels from (at least in part) pre-verified modeling constraints. Such composition begins by selecting predefined and pre-validated general constraints from a constraints library, as discussed above, then applying the necessary domain-specific concepts and constraints. Experience has shown the general modeling composition constraints to be larger and more complex than domain-specific constraints. Therefore, reuse of these general composition constraints significantly reduces the time required to create metamodels. Also, because the general modeling composition constraints have been validated *a priori*, the metamodeler can concentrate on the domain-specific aspects of semantic specification. The metamodel composition process is shown in Figure 4.

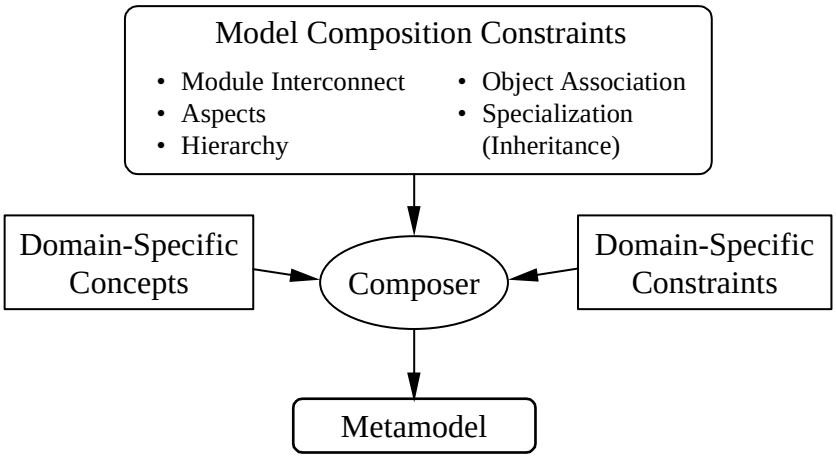


Figure 4: Metamodel composition

Once the syntactic and semantic specifications for a modeling language are composed into a metamodel, a modeling language specification exists, albeit still abstract. Additional specifications regarding how the modeling environment presents the language's entities and relationships to the modeler must be made. In other words, the language specification is not a specification *for an entire modeling environment*. It can be argued that presentation specifications are merely additional syntactic specifications. As mentioned earlier, however, for reasons of portability and interoperability it is desirable to keep the presentation specifications separate from the syntactic and semantic specifications.

Finally, since a general modeling environment should include facilities for extracting information from model instances created using the environment, a set of model interpretation specifications should be included when specifying a complete DSME. Such interpreter specifications are a form of semantic specification, but as with the presentation specifications, it is better to develop and maintain interpreter specifications separately from the syntactic, semantic, and presentation specifications already discussed. See [9] for a discussion of the theory and practice of specifying interpreter behavior.

As mentioned earlier, synthesizing and evolving DSMEs is done by using the metamodel to configure a general modeling environment for use within a particular domain.

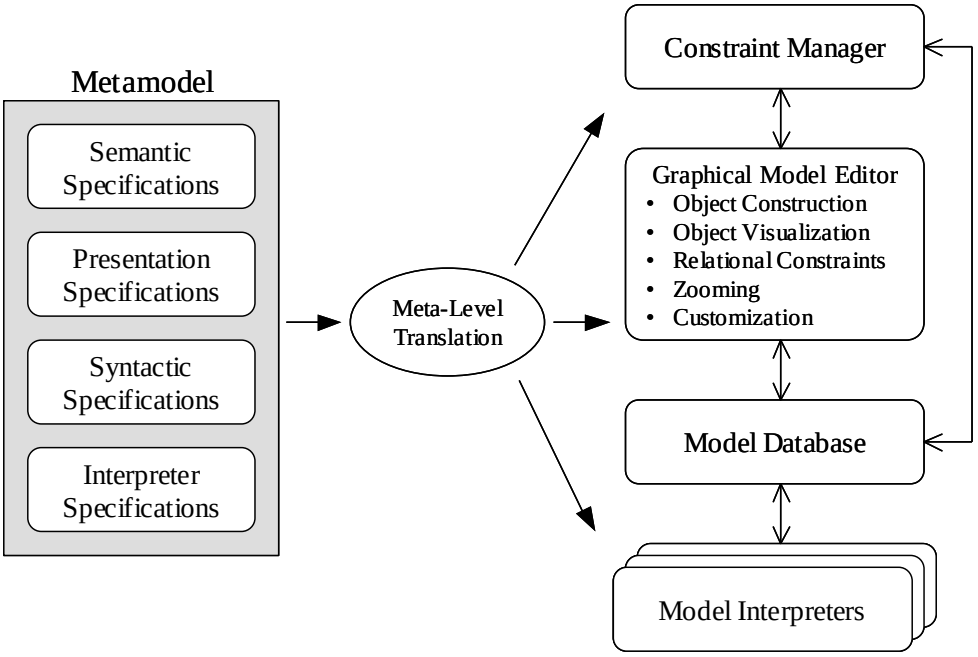


Figure 5: Metamodel translation

Figure 5 shows a metamodel on the left containing the various specifications previously discussed. Translating the various metamodel specifications into a form suitable for configuring the DSME is done by the meta-level translator. On the right are the configurable components of a general DSME. The Constraint Manager, which is responsible for ensuring that only valid models are created in the target domain, is configured using information contained in the metamodel's semantic specification (i.e. constraint equations). The Graphical Model Editor is configured by combining information from the semantic, presentation, and syntactic specifications. This includes managing how various aspects of the models are presented, how objects are created, controlling the type and multiplicity of object associations, as well as allowing storage and retrieval of models from persistent storage. Model Interpreters are partially configured using information from the metamodel's interpreter specification.

C. Constraint Management

The activity of modeling is essentially choosing a particular model from an infinite set of possible models. By limiting the types of modeling objects and relationships allowed in the models, the set of possible models can be greatly reduced (of course, the set can still be infinite!) As discussed in the previous section, these limitations represent the static semantics of a modeling language, and, as such, appear as domain-specific modeling constraints in the metamodel. Of course, such constraints can only be enforced in the presence of actual domain-specific models – i.e. model *instances* created using the modeling language specified by the metamodel. Enforcing these constraints is done by a *constraint manager*. The constraint manager is a configurable component of the DSME. It provides various cues to the modeler according to the static semantics described in the metamodel.

Consider again the aircraft in-flight safety system modeling environment example. It was stated earlier that sensors can be connected to actuators, and vice versa. By specifying the entities (actuators and sensors), the connection roles and multiplicities, and a simple mapping to particular

graphical objects, the resulting sensor-to-actuator connection specification could be easily enforced by a graphical model editor – the graphical editor would only allow interconnections between actuators and sensors, and disallow all other types of connections (e.g. actuator-to-actuator, sensor-to-sensor, etc.) However, suppose a domain-specific constraint is included in the metamodel stating that every sensor must be connected to *something* (a very important real-world consideration). Although such a constraint can easily be stated in the metamodel (for example, using a UML multiplicity specification or by including an invariant expression stating that the size of the set of actuator objects connected to the output of any sensor be greater than zero), such a constraint can only be checked once a specific model exists. In other words, the graphical model editor can *prevent* certain editing actions, but cannot *guarantee* certain editing actions. And although the constraint manager cannot guarantee certain editing actions either, it can indicate whether, at a given point in time, a certain model does or does not satisfy a particular constraint. Such a constraint enforcement mechanism should not be confused with any formal metamodel validation performed before synthesizing a DSME from a metamodel. Metamodel validation ensures that any modeling environment created from the metamodel specifications will generate valid models in the domain (i.e. that the modeling language specification is *consistent*).

IV. Using the Metamodeling System

To illustrate the process of specifying and synthesizing a DSME using the MGA metamodeling environment, a MIPS environment for building custom process monitoring and simulation applications for chemical plants will be presented. Before discussing the actual metamodel itself, however, it is necessary to discuss the various components of the MGA graphical modeling environment.

A. The GME Modeling Environment

Figure 6 illustrates the MGA MIPS environment architecture. At the core is the Graphical Model Editor (GME). This component is responsible for maintaining the model structures and providing the operations to manipulate them. The GME communicates with the other components through the Component Object Model (COM) [10].

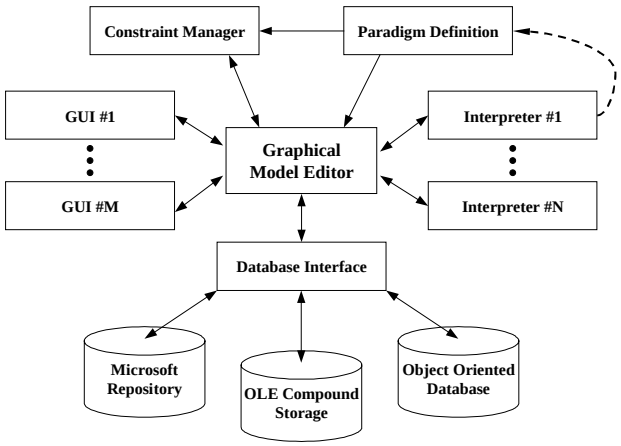


Figure 6: MGA MIPS Environment Architecture

Different graphical user interfaces are allowed to access the models. Our own graphical interface supports all the idioms and operations provided by the GME. A table editor is also available that speeds up the entry of primarily textual information. A text-based interface is supported, but it is inherently more difficult to use for complex paradigms and/or models. Interfacing COTS drawing packages to GME is also possible. For a given tool (e.g. Visio), a simple layer needs to be implemented that maps the GME COM interface to that of the COTS package. Depending on the tool, some graphical idioms or operations may not be accessible this way.

Model storage is provided transparently by the database interface to Microsoft Repository [11] or object oriented databases. Though not strictly a database, OLE compound storage provides the basic functionality required to store models. For simpler paradigms and small- to medium-sized models, this is a satisfactory solution. It also provides the benefit of not having to install a large, complex third party database package.

The Constraint Manager has access to the set of domain-specific constraints provided by the Paradigm Definition module and the models maintained by the GME. Checking of selected constraints can be triggered by certain requested operations, such as *connect*, *close*, or *modify attribute*. All constraints can be checked at once upon demand. Constraints are specified using the MultiGraph Constraint Language (MCL), an extension of the standard Object Constraint Language (OCL) [12][13].

The Paradigm Definition module contains configuration information for the graphical idioms and operations that constitute a given paradigm. It is directly generated by the metaprogramming layer of the MGA. Since that layer is implemented by the same MIPS infrastructure, Figure 6 can be interpreted as the metaprogramming environment for generating domain-specific MIPS environments. If the models describe the metaprogramming environment (i.e. in case of the meta-metamodels) then the toolkit configures itself (i.e. the toolkit is *self-describing*). The dashed line from one of the model interpreters to the paradigm definition module illustrates this concept.

The interpreter interface enables model interpreters to be written in any language that supports COM. The interface allows full access to the models, either to extract information from them or to modify the models themselves. Visualization hooks are also provided enabling the interpreter to generate meaningful visual feedback in the correct context to the user.

An interpreter can be implemented as a dynamic link library (DLL) or as an executable module. Using DLLs will result in in-process activation of the interpreter, and consequently, negligible overhead. The drawback is that they are not independent standalone modules – the user needs to launch them from the GME. Executable interpreters have the flexibility of being able to initiate interpretation independently from the GME. However, they run considerably slower because of the context switch performed by the operating system at every interface function call.

Interpreter DLLs are registered in the Windows registry on a per paradigm basis. The actual DLL is not loaded until the user requests interpretation from the GME and selects the desired interpreter (selection is necessary only when multiple interpreters have been registered for the same modeling paradigm).

GME also supports event-based interpreters, called *add-ons*. There is a set of predefined events, such as *create model*, *disconnect*, *modify attribute*, etc., associated with add-ons. Add-ons can register to respond to any or all of these events. When an event occurs, the GME passes control to the registered add-ons one-by-one. Add-on code then has full access to the models through the COM interpreter interface.

Add-ons can be either paradigm-independent or paradigm-specific. Thus, the services of the GME can be extended using add-ons in general, or specifically for a given paradigm. For example, if a modeling paradigm states that the names of two connected objects within a given kind of model must be the same, a small Visual Basic script can be written to enforce this requirement.

Add-ons and event-based constraints can both be used to restrict the user's ability to create invalid models. However, add-ons extend this capability by providing a mechanism (via code) to rectify the situation. Furthermore, in some cases a constraint is impossible to specify in MCL, and the expressive power of a programming language is needed (e.g. ensuring a hierarchical graph does not have cycles). Add-ons are not part of the paradigm definition (i.e. not defined in metamodels), and because they use COM-aware high-level programming languages, they are not formally defined.

Despite its flexibility, the COM interface is still a low-level interface. For all but very simple interpreters, the interpreter writer must create complex data structures and traverse the model hierarchy several times to properly initialize data members. Our experience has shown that much of the low-level code required to traverse these hierarchies is similar in nature (e.g. loop constructs for traversing data graphs, etc.) irrespective of the individual modeling paradigms. We have implemented this common functionality in the form of an extensible, high-level C++ interpreter interface existing as a layer above the COM interface, hiding the intricacies of COM programming (Figure 7).

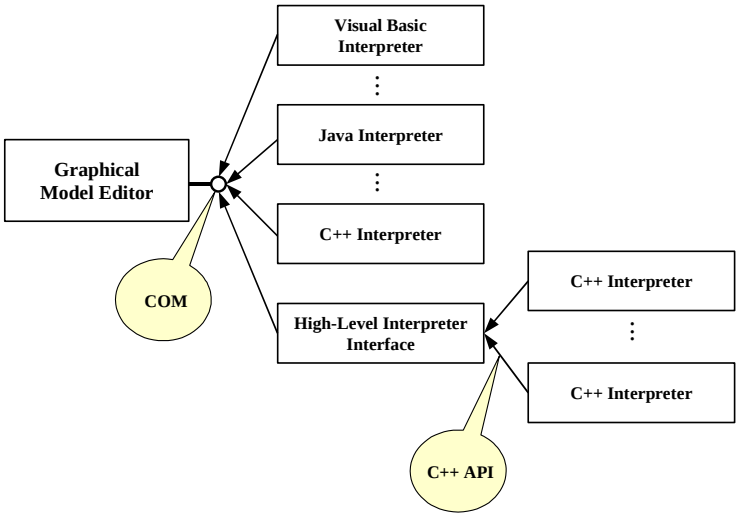


Figure 7: The GME Interpreter Interface Architecture

This interpreter interface defines its own generic classes and builds up a *builder object network* before the user-defined interpreter receives control. This builder object structure closely resembles that of the models. However, the builder objects facilitate traversing the models along the hierarchy, connections, or object references more efficiently than using the actual model data structures themselves. For example, using the builder object network it is possible to find connected, deeply nested leaf nodes using a single function call.

The interpreter interface is easily extensible: the builder classes can be extended with inheritance and the interpreter interface automatically instantiates the user-defined classes though the use of an object factory.

The GME, its native graphical user interface, the Constraint Manager, the database interface, and the interpreter interface are all paradigm-independent modules that configure themselves at runtime. This is in contrast to earlier versions of our tools, where the components were paradigm-specific and automatically generated in their entirety from metamodels [15]. Our approach speeds up the design cycle of creating and evolving domain-specific MIPS environments [16]. The metamodeler can edit the metamodels, generate the DSME specification with the appropriate metamodeling interpreter, and without exiting the metamodeling environment load the generated DSME and begin building models. Designing a modeling paradigm is an inherently iterative process, so speeding up this cycle can result in significant productivity increases [16].

B. The Activity Modeling Tool

As an example of specifying and synthesizing a DSME from a metamodel, we present the Activity Modeling Tool (AMT). The AMT is a MIPS environment for building custom process monitoring and simulation applications for chemical plants. AMT provides a means to model all the components necessary to (1) interface to a real-time database that gathering plant sensory data, (2) define custom processing steps, (3) create an operator interface, and (4) interface to a COTS process

simulator. From this set of integrated models, the interpreters synthesize the components and combine them together to form a custom process monitoring and simulation application.

Signal models are the most common building blocks in this domain. Signals represent data that flow between components. Signal models provide the connection from the real-time database interface to the custom processing modules that are captured using a hierarchical signal flow representation. The operator interface and simulator interface models also use signals to capture their input and output data.

Each signal is associated with a data type. This can be a simple built-in type such as a double, float, integer, or character; an array of any one of the simple types; or a user-defined aggregate type. Aggregate types are explicitly modeled by combining simple (and array) built-in types.

The entire AMT paradigm is quite complex, and a full explanation is beyond the scope of this article. Here we shall focus only on the hierarchical signal flow models extended with data types. Figure 8 shows the corresponding partial metamodel.

In the GME metamodeling environment, the syntax and part of the static semantics of the domain are captured using UML class diagrams. In the AMT paradigm, the *Compound*, *Primitive*, *InputSignal* and *OutputSignal* classes are the basic building blocks of hierarchical signal flow models. The *Processing* and *Signal* classes are abstract base classes that are used to simplify the metamodel itself. *Primitive*s are elementary objects that correspond to computations at the subroutine level in the target domain. The subroutine itself is specified using a script attribute. In the context of the signal flow models, *Primitive*s can contain *Input*- and *OutputSignal*s (through the aggregation inherited from the *Processing* class), but not *Processing*s. On the other hand, *Compound*s can contain *Processing*s, i.e. *Primitive*s and *Compound*s, creating a hierarchy of possibly unlimited depth. *DataflowConn* is an association among *Signal*s. Its name indicates that this association is implemented using GME connections as specified in the presentation aspect of the metamodel (not shown in Figure 8).

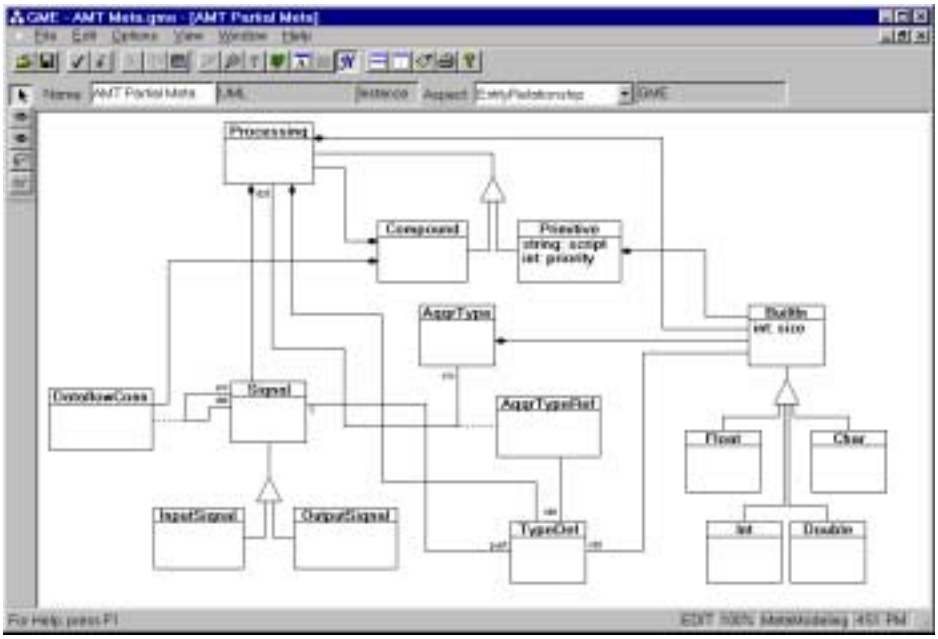


Figure 8: Partial metamodel of the AMT environment

The *DataflowConn* association allows connections between two *Signal*s. Because *Signal*s are specialized into *InputSignal*s and *OutputSignal*s, *DataflowConn* does not prohibit connecting the *OutputSignal*s of two *Primitive*s together. (Of course, a more specific form of *DataflowConn* could have been created limiting connections between *InputSignal*s and *OutputSignal*s, but this approach is generally too restrictive, and the more flexible module interconnection [17] approach shown above is preferred). Thus, it is necessary to augment the UML-based static semantics of the AMT modeling language by specifying explicit constraints in OCL (extended by MGA-specific constructs). We add a constraint to our metamodel that prohibits *OutputSignal* to *OutputSignal* connections:

```
connections("DataflowConn")->forAll(c |
      c.source().kindOf() = "OutputSignal" implies
            c.destination().kindOf() <> "OutputSignal");
```

Here, `connections()`, `source()`, `destintion()`, and `kindOf()` represent MGA-specific extensions to OCL. They are "functions" that provide access to different GME entities and relationships in the models. The above statement means that *DataflowConn* connections that originate from an *OutputSignal* cannot terminate at an *OutputSignal*. (*NB:* The constraint given above is somewhat simplified for illustrative purposes. In this particular paradigm, depending on the hierarchical placement of the source and destination *OutputSignal* objects, there are cases when connecting two *OutputSignals* is necessary and must be allowed, resulting in a more complicated constraint equation.)

The remaining classes in Figure 8 correspond to data type specification for *Signals*. The *BuiltIn* class is an abstract base class for the four supported simple types: *Float*, *Double*, *Int* and *Char*. The size attribute specifies whether a particular simple type instance is a single variable or an array. *AggrType*s consist of *BuiltIn* types. *AggrType*s are created separately from signal flow models, i.e. *Processing*s do not contain *AggrType*s, because *AggrType*s are used in other kinds of models (not described here). However, for data type specification, *Signal*s contained in *Processing*s need to be associated with *AggrType*s. As shown in the UML diagram, the AMT paradigm allows an *AggrTypeRef* association between *Processing*s and *AggrType*s. As its name indicates, this association is implemented using the MGA Reference construct. A Reference is similar to a pointer; it is not an actual model – it merely refers to one. For type specification purposes, *Processing*s can contain references to *AggrType*s as well as instances of *BuiltIn* types.

*AggrTypeRef*s, in turn, can participate in *TypeDef* associations. *TypeDef* is an association between a single type model (*AggrType* or *BuiltIn*) and a set of *Input*- and/or *OutputSignals*. This is implemented using the MGA construct called Conditional (described in [14]).

After the syntactic and semantic information have been captured in a metamodel, the presentation specification must be added. This step is very specific to the MGA and is beyond the scope of this paper, but basically represents a mapping between UML class objects and associations and GME models representing the various GME modeling resources (e.g. Containment, Connections, References, etc.). Once the metamodel is complete, a meta-level translator generates the necessary configuration files to configure a new domain-specific MGA modeling environment.

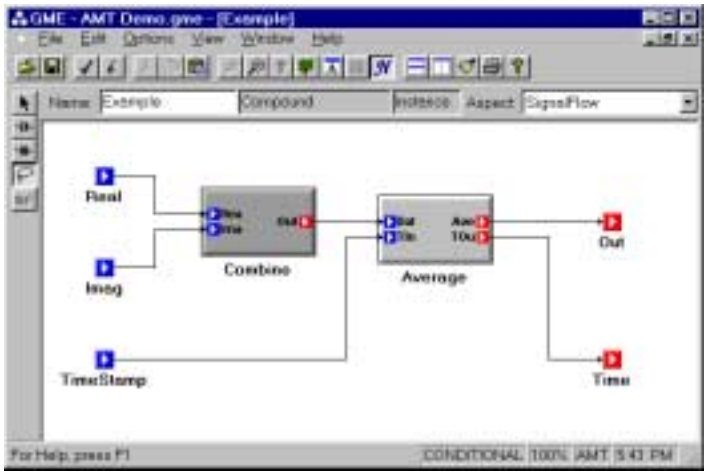


Figure 9: SignalFlow Aspect of Compound Model "Example"

In this new environment, the signal flow and the data type specifications are shown in different aspects (as specified by the presentation specifications in the metamodels).

Figure 9 depicts the SignalFlow aspect of an example signal flow model. This simple signal flow model shows a Compound-type model with three *InputSignal*s and two *OutputSignal*s. The "Combine" model is a Primitive (cannot contain other models), while "Average" is another Compound. The connections in the figure are all *DataFlowConn*s. Type specification for all the *InputSignal*s and *OutputSignal*s can be found in the SignalTypes aspect of the same model depicted in Figure 10.

Figure 10: SignalType aspect of Compound Model "Example"

The association between signal type objects and signals ("*TypeDef*" in Figure 8) is hard to visualize in a single picture because at any one time only one instance of such an association can be shown. In Figure 10, the selected atom Double is associated with all the objects that are highlighted (as opposed to grayed out) – in this case the InputSignals "Real" and "Imag." In addition to the BuiltIn objects (a Double and an Int), the figure contains a reference to a user defined AggrType called

ComplexRef. In this example, ComplexRef is associated with the OutputSignal "Out," but this is not shown in the figure.

The correctness of the signal type specification is enforced using a set of metamodel-level constraints. The GME Constraint Manager component uses these constraints to ensure that every InputSignal and OutputSignal is associated with exactly one type (simple or aggregate), and that signals at each end of a connection are of the same type and have the same “size” attribute.

V. Summary of Metamodeling Languages

Metamodeling is currently an active research topic. Many metamodeling languages and environments are being developed, each with its own set of capabilities and restrictions. To position the research discussed in this paper within the field, a review of these languages and environments was conducted. The criteria for comparison is completely described in [16], and is summarized below.

- Support for the four-layer metamodeling architecture.
- Ability to model both the abstract syntax and static semantics of a modeling language.
- Ability to separate modeling language syntax and semantic specifications (the "function") from implementation details (the "form").
- Ability to compose metamodels from pre-specified, generalized modeling constraints, supplemented with necessary domain-specific concepts and constraints.
- Ability to specify static semantics as a set of provably correct invariant expressions (i.e. *constraint* expressions).
- Ability to validate the consistency of metamodel using machine-aided methods such as theorem provers or proof checkers.
- Support for metamodeling tool interoperability by allowing metamodels to be translated to and from other metamodeling languages.

The following table lists the languages surveyed, along with the criteria for comparison. (*NB*: This table has been modified from the table in [16] to reflect the current capabilities of the MGA-based metamodeling environment), and to include ProtoDOME [20].

Table 3: Comparison of metamodeling languages

Language: / Capability:	Aesop, Armani	AML	CDIF	DF	Express	MGA	MOF	ProtoDOME	Larch, Specware
Four-layer support			X			X	X	X	
Abstract syntax modeling	X	X	X	X	X	X	X	X	X
Static semantic modeling	X	X	X	X	X	X	X	X	X
Metamodel composition									X
Constraint language		X			X	X			X
Proof checker									X
Metamodeling tool interoperability		X	X		X	limited	X	limited	

VI. Conclusions and future work

This paper has presented a method for formally representing DSME requirements as metamodels using UML class diagrams and predicate logic constraint language expressions which are then used to synthesize the target DSME itself. The DSME is then used to create models of domain-specific systems. Model interpreters are used to apply semantic meaning to the models. This semantic information is used as a basis for model analysis and in translating the domain models into executable models (e.g. domain-specific applications) or into data streams intended for use by third-party analysis and/or execution environments.

This approach has three distinct advantages. First, by formalizing the specification process, DSME design requirements can be stated hierarchically, using information as it becomes available, allowing DSME specifications to be refined as the design process proceeds. Also, the formalized specifications improve communication and information exchange among customers and developers. Second, once the design has been finalized (i.e. once the metamodel is complete), synthesis of the target DSME from the metamodel is rapid and less error prone than previous manual implementation methods. Finally, this approach allows DSMEs to be evolved in a safe and controlled manner as domain modeling requirements change.

Current and future research in DSME specification and generation centers around two goals: making better use of the existing metamodel design formalisms (e.g. UML and MCL), and allowing interpreter specifications to be stated formally in the metamodel. It is anticipated that libraries of metamodel solutions to “standard” modeling problems will be developed and incorporated into the metamodeling environment (e.g. standardized representations for module interconnection schemes or for representing aspects in a target DSME). Also, by formally specifying and including key model interpreter features into metamodels, much of the interpreter code currently written by hand can be generated as part of the DSME synthesis process.

References

[1] J. Sztipanovits, “Engineering of Computer-Based Systems: An Emerging Discipline,” *Proceedings of the IEEE ECBS’98 Conference*, 1998.

[2] J. Sztipanovits, et al.: “MULTIGRAPH: An Architecture for Model-Integrated Computing,” *Proceedings of the IEEE ICECCS’95*, pp. 361-368, Nov. 1995.

[3] D. Oliver, T. Kelliher, J. Keegan, Jr., *Engineering Complex Systems with Models and Objects*. New York: McGraw-Hill, 1997.

[4] S. White, et al.: “Systems Engineering of Computer-Based Systems", *IEEE Computer*, pp. 54-65, November 1993.

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

[6] J. Ernst: "Introduction to CDIF", http://www.eigroup.org/cdif/intro.html.

[7] *UML Semantics*, ver. 1.1, Rational Software Corporation, et al., September 1997.

[8] Rice, M.D. and Seidman, S.B.: “A Formal Model for Module Interconnection Languages”, *IEEE Transactions on Software Engineering*, Vol. 20, No. 1, pp. 88-101, Jan. 1994.

[9] G. Karsai, et al., “Towards Specification of Program Synthesis in Model-Integrated Computing," *Proceedings of the IEEE ECBS’98 Conference*, 1998.

[10] D. Box, *Essential COM*, Addison-Wesley 1998.

[11] Microsoft Repository, http://msdn.microsoft.com/repository/default.asp, Microsoft Corp., 1999

[12] *Object Constraint Language Specification*, ver. 1.1, Rational Software Corporation, et al., Sept. 1997.

[13] J. Warmer, A. Kleppe, *The Object Constraint Language*. Addison-Wesley 1999.

[14] A. Ledeczi, et al., “Metaprogrammable Toolkit for Model-Integrated Computing," *Proceedings of the IEEE ECBS’99 Conference*, 1999.

[15] G. Nordstrom, J. Sztipanovits, G. Karsai, “Meta-level Extension of the Multigraph Architecture," *Proceedings of the IEEE ECBS’98 Conference*, 1998.

[16] G. Nordstrom, “Metamodeling – Rapid Design and Evolution of Domain-specific Modeling Environments," Doctoral Dissertation, Vanderbilt University, 1999.

[17] M. Rice and S. Seidman, “A Formal Model for Module Interconnection Languages,” IEEE Transactions on Software Engineering, Vol. 20, No. 1, pp. 88-101, Jan. 1994.

[18] CORBA web site, http://www.corba.org/, OMG, 2000

[19] POSIX web site, http://anubis.dkuug.dk/JTC1/SC22/WG15/, DKUUG, 2000

[20] DOME web site, http://www.src.honeywell.com/dome/, Honeywell, 2000