Metamodeling – Rapid Design and Evolution of Domain-Specific Modeling Environments

Greg Nordstrom, Janos Sztipanovits, Gabor Karsai, and Akos Ledeczi
Institute for Software Integrated Systems
Vanderbilt University

Abstract

Model integrated computing (MIC) is gaining increased attention as an effective and efficient method for developing, maintaining, and evolving large-scale, domain-specific software applications for computer-based systems. MIC is a model-based approach to software development, allowing the synthesis of application programs from models created using customized, domain-specific model integrated program synthesis (MIPS) environments. Until now, these MIPS environments have been handcrafted. Analysis has shown that it is possible to "model the modeling environment" by creating a metamodel that specifies both the syntactic and semantic behavior of the desired domain-specific MIPS environment (DSME). Such a metamodel could then be used to synthesize the DSME itself, allowing the entire design environment to safely and efficiently evolve in the face of changing domain requirements. This paper discusses the use of the Unified Modeling Language and the Object Constraint Language to specify such metamodels, and describes a method for incorporating these metamodels into the MultiGraph Architecture, a MIPS creation toolset.

Background

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 a CBS system's life cycle, changes in mission requirements, personnel, hardware, support systems, etc., all drive changes to the CBS. Reconfiguration via software has long been seen as a potential means to effect rapid change in such systems. An emerging technology that enables such system evolution is model integrated computing (MIC).

Model-integrated computing

MIC is a methodology for generating application programs automatically from multi-aspect models. One approach to MIC is Model Integrated Program Synthesis (MIPS). MIPS allows experts in a particular domain to create an integrated set of models representing all or part of various domain-specific systems. The models are then used for system analysis or as a source from which to automatically generate executable models (i.e. executable programs) which run on the actual system. The MultiGraph Architecture [2], under development at Vanderbilt University, is a toolkit for creating MIPS environments.

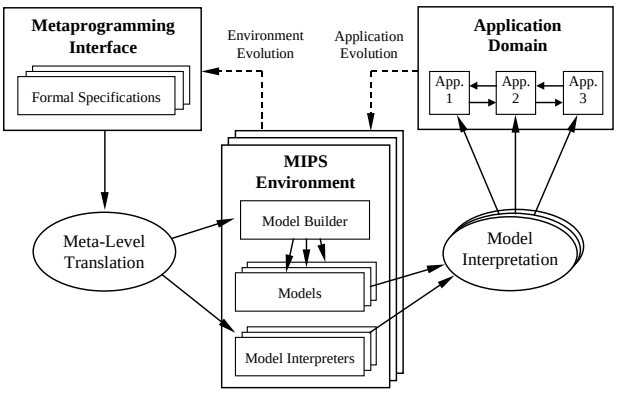


Figure 1: The MultiGraph Architecture

A MIPS environment operates according to a *modeling paradigm*. A modeling paradigm is a set of requirements that governs how systems within the domain are to be modeled. The modeling paradigm defines the *family* of models that can be created using the MIPS environment. Said another way, the modeling paradigm defines the language for modeling systems in the domain. The modeling paradigm is captured in the form of formal modeling language specifications called a *metamodel*.

Once a domain-specific modeling language has been formally defined, a meta-level translation can be performed to synthesize the domain-specific MIPS environment (DSME) from the metamodel. The DSME is then used by domain experts to create various models of domain-specific systems. Once one or more domain models exists, *model interpreters* are used to perform semantic translations on the models in order to generate executable models or perform various types of data translation and analysis.

Metamodeling

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 or processes, and when the process being modeled is *the process of 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 and is in general use by the metamodeling community. The following table, taken from [3], describes each layer of this framework:

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.

Table 1: Four-layer metamodeling architecture

This four-layer metamodeling 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 [3].

To properly specify a modeling language using a metamodel, the *syntax* and *semantics* of the language must be modeled.

Modeling syntax

To capture the syntax of a modeling language, a metamodel must describe all the entities and relationships that may exist in the target language. As discussed in [3], 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. In a graphical metamodel, the syntax is modeled as a collection of modeling object types, along with all relationships allowed between those object types.

As an example, consider a DSME for embedded processor modeling. A metamodel describing such an environment would likely include `processor` and `sensor` entities, and a `connectedTo` relationship specifying an allowed association between sensors and processors.

Modeling semantics

In addition to specifying the syntax of a modeling language, a metamodel must specify the semantics. Continuing the embedded processor example, in addition to specifying the types of entities involved in the `connectedTo` relationship (a syntactic specification), the metamodel must specify the allowable *number* of individual processors and sensors that can participate in the `connectedTo` relationship (a semantic specification). So the metamodel must specify the multiplicity of the `connectedTo` relationship. Depending on the particular types of processors and sensors selected, such a metamodel might specify that one sensor can be connected to at most three processors, or that one processor can be connected to no more than five sensors.

At this point 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 refer to the interpretation of a given set of modeling constructs *in the context of the model instances themselves*. Only the static semantics may be specified in a metamodel, since the metamodel has no way of knowing *a priori* what meaning to associate with particular instances (i.e. particular models)

created using the language. Distinguishing between static and dynamic semantics is best illustrated by an example.

Consider a DSME used to model the behavior of real-time scheduling systems. A metamodel description of such a DSME 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. Scheduler models that violate any of these constraints are, by definition, invalid models. Two possible constraints, stated using standard English, might be:

- "Task duration must be specified"
- "Schedules can hold a maximum of 10 tasks"

Such constraints represent the static semantics of the modeling language, and can be defined in the metamodel. However, the following constraints represent dynamic semantics, and as such, cannot be represented as metamodel constraints:

- "All tasks must meet their execution deadlines"
- "Scheduling queues must never overflow"

The metamodel can define what it means for a task to have a duration associated with it (e.g. by requiring certain values for a “duration” attribute), and can specify that such a "meaning" must be satisfied in any model that includes tasks. If the modeler fails to provide a value for task duration, the model is considered invalid. However, the metamodel cannot specify task schedulability, since schedulability is a function of, among other things, certain run-time factors – factors that the metamodel has no *a priori* knowledge of. Unless otherwise indicated, references to modeling language “semantics” in the remainder of this paper refer to “static semantics.”

Closely associated with specifying the semantics of a modeling language is the form that the constraint expressions take. Constraints must be stated in such a way as to be precise and analyzable, so that before any modeling language or modeling environment is synthesized from a metamodel, it can be shown (i.e. *proved*) that the metamodel itself is semantically consistent (i.e. the constraints do not contradict each other or form a constraint set that can never be satisfied). Therefore, the static semantics should be specified in a mathematical language. An ideal candidate is any first- or higher-order predicate calculus, since the invariants can take the form of Boolean expressions that must be satisfied by any model created using the target modeling language. Such expressions can be tested using a proof checker before the metamodel is used to generate a modeling language.

Presentation specifications

When a metamodel is used to specify a graphical modeling language, it is important to separate presentation specifications (i.e. specifications concerned with how objects should appear on screen, how entities and relationships are viewed in different aspects, etc.) from the syntactic and semantic specifications. While such presentation specifications are necessary to implement the target DSME, the presentation specifications do not add meaning to the simple and precise syntactic and semantic modeling language specifications, and will, oftentimes, obscure those specification, making metamodel interpretation more difficult.

Making modeling (and metamodeling) tools interoperable requires that metamodels be exchanged among various metamodeling tool suites. This requires the modeling language's syntax and semantics be precisely specified by the metamodel, even if other aspects, such as implementation and presentation details, are not. This underscores the need to separate presentation specifications from semantic and syntactic specifications.

Interpreter specifications

As mentioned earlier, model interpreters are used to perform semantic translations on the domain models created using DSMEs. Until now, these model interpreters have been hand-crafted in C++. However, research is being conducted into ways of specifying interpreter behavior in metamodels [4]. Such behavioral specifications would then be used to synthesize all or part of the necessary domain-specific model interpreters. Although interpreter specification is beyond the scope of this paper, is should be noted that interpreter specifications, like presentation specifications, should be cleanly separated from syntactic and semantic specifications.

In summary, when considering a metamodeling language for the MGA, several requirements exist, as listed 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 from presentation and/or interpreter specifications.

- Ability to compose metamodels from pre-specified, generalized modeling constraints, supplemented with necessary domain-specific concepts and constraints. (See [5] for an in-depth discussion of metamodel composition).
- 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.

Environment generation

Once a metamodel has been created, it is used to synthesize and/or configure various components in the DSME. Because the metamodel contains syntactic, semantic, presentation, and interpreter specifications, most, if not all, of the DSME can be directly synthesized from the metamodel.

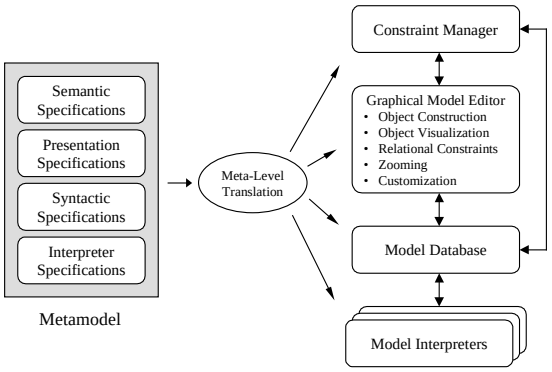


Figure 2: Metamodel Translation

Figure 2 shows that by performing a meta-level translation, the metamodel specification on the left can be used to generate the DSME on the right. The semantic specifications in the metamodel are used by the DSME constraint manager to verify that models created using the DSME are legal – i.e. they don't violate any of the semantic constraints specified in the metamodel.

The presentation and syntactic specifications are used to configure the DSME's graphical model editor. 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.

Information in the metamodel is also used to generate the schema for the model database and the database interface code for the constraint manager and the graphical model builder.

Finally, as discussed in [4], portions of the model interpreters can also be synthesized from the interpreter specifications contained in the metamodel.

UML- and OCL-based metamodeling

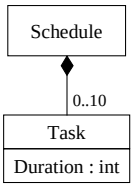
The Unified Modeling Language (UML), along with the Object Constraint Language (OCL), have been adopted for use in defining MGA metamodels. Together, UML and OCL meet all the metamodeling language requirements listed above.

UML is an OMG-approved graphical modeling language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system [6][7]. It combines concepts from the Booch Method, Rumbaugh's Object Modeling Technique, and Jacobson's Object Oriented Software Engineering method. UML supports many modeling notions, such as use-case diagrams, class diagrams, implementation diagrams, and behavior diagrams (including state charts, state machines, activity diagrams, sequence charts, and collaboration diagrams). UML is a specification language, and as such it does not cover tool specifications, diagram layouts, coloring, user navigation, and other presentation issues.

UML supports the four-layer metamodeling architecture, and can be used to model other modeling languages. When used to model an MGA modeling language, the syntax requirements of the modeling language are captured in the form of graphical, entity-relationship diagrams using UML *class diagrams*. Modeling language semantics are represented as invariant predicate logic expressions using the Object Constraint Language (OCL)[8]. OCL is a public domain, formal language specification that can be used to express modeling language constraints and other expressions associated with graphical models. OCL is a textual language, designed to be used in conjunction with, but independent of, UML. The full OCL specification can be found in [8]. Although OCL is formal and has an exact syntax, it was also designed to be easily read and understood by human designers. OCL is a typed language, and OCL expressions to be type conformant (e.g. a designer cannot compare string values with Boolean values). Enumerated types are also supported in OCL.

OCL is not a programming language, and OCL constraint expressions cannot directly affect any models created using the target modeling language. The constraint expressions are merely formal comments on the semantics of the modeling language. Models created using the target modeling language can be verified using the OCL expressions, but the expressions cannot cause any changes in the models.

In an MGA metamodel, OCL statements represent invariant Boolean expressions that specify the semantics of the target modeling language. As an example, in the scheduling system previously discussed, it was mentioned that schedules can hold a maximum of 10 tasks. Such a requirement can be easily specified using UML class diagrams as follows:



This diagram states that a `Schedule` object can contain from zero to 10 `Task` objects. However, there was another requirement that the duration of each task must be specified. Because no graphical mechanism exists in UML to specify such a requirement, OCL must be used. The following OCL expression specifies this requirement:

```
Task.allInstances->forAll(t |
    t.Duration > 0)
```

This expression states that the `Duration` attribute of every `Task` instance must have a value greater than zero. Such an expression represents a semantic constraint in the particular modeling paradigm.

Both UML and OCL are *de facto* industry modeling standards, developed by a large consortium of industry leaders. They enjoy wide popularity in the modeling community and have been used in a wide variety of modeling applications. Public domain parsers exist for checking OCL specifications, and several public domain and commercial UML software development environments are available.

Proof-of-concept example

To demonstrate the use of UML/OCL in specifying an MGA modeling paradigm, consider an audio processing system consisting of microphones, preamplifiers, power amplifiers, and speakers. Each of these modeling objects can be represented using a set of UML class objects. A UML class object describes a set of objects sharing a set of features, such as attributes, operations, methods, relationships and semantics. (MGA UML-based metamodels currently do not use class operations or methods.)

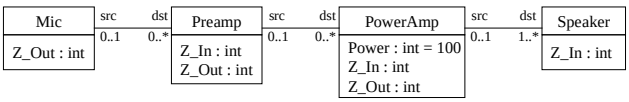


Figure 3: Simple UML Audio Processing Metamodel

Figure 3 shows a simple UML metamodel representing this audio signal processing modeling paradigm. The metamodel specifies the types of modeling objects allowed (e.g. `Mic`, `Preamp`, etc.), object attributes (e.g. `Z_In`, `Z_Out`, etc.), as well the associations allowed among the objects (e.g. every `Mic` object, playing the role of `src`, may be associated with zero or more `Preamp` objects, playing the role of `dst`. Similarly, every `Preamp`, playing the role of `dst`, may be associated with zero or one `Mic`s playing the role of `src`). Note that the `Power` attribute of each `PowerAmp` will be initialized to 100 when `PowerAmp` objects are instantiated (this value may later be changed by the modeler). Also, the association between `PowerAmp` objects and `Speaker` objects requires every `PowerAmp` to be associated with *at least one* `Speaker`. This requirement means that any audio processing system model in which a `PowerAmp` is not connected to a `Speaker` would be illegal.

Although not shown in Figure 3, there is another requirement of this modeling paradigm – every audio processing model must contain at least one `PowerAmp`. Such a requirement cannot be stated using UML class diagrams alone, since the presence of a `PowerAmp` class diagram in this metamodel merely states that `PowerAmp`s are *allowed* in audio processing models – there is no way to indicate graphically that a certain number of these objects are required. That requirement must be stated using an OCL constraint as follows:

```
PowerAmp.allInstances->size >= 1
```

This states that all audio processing models must contain at least one `PowerAmp` instance to be valid. (*NB*: OCL expressions can be included in UML diagrams using

the UML *textbox* notation, but are shown here as free-standing equations for convenience. In any case, the OCL expressions are considered part of the metamodel.) Note that while modeling language semantic requirements such as this are stated in the context of a metamodel (i.e. at metamodeling time), they are used to verify model instances created using the DSME at model building time.

The metamodel of Figure 3 represents a fairly brittle specification of the audio processing modeling language. Although every modeling object contains one or both of the impedance attributes `Z_Out` and `Z_In`, hierarchical decomposition has not been used to create specialized modeling objects from more general objects. Also, associations between objects are made at the lowest possible level – between the components themselves – instead of between hierarchically more general objects. And while there is an underlying relationship between the impedance values and the association roles (e.g. the `Z_In` attribute of the `PowerAmp` is related to the `dst` role of the association between the `Preamp` and the `PowerAmp`), the metamodel indicates no such direct relationship.

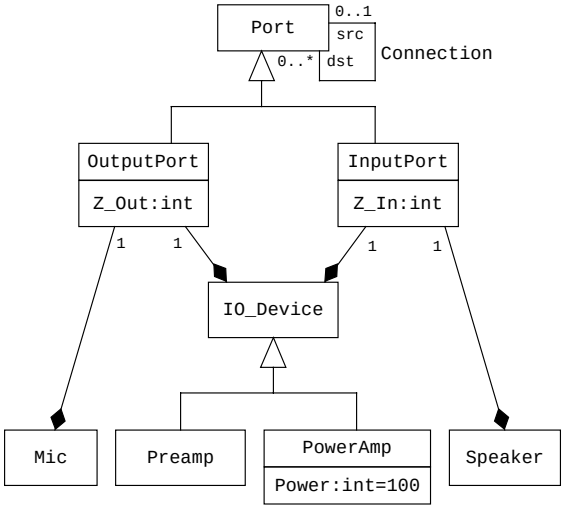


Figure 4: Refined Audio Processing Metamodel

Figure 4 shows a refined audio processing metamodel using object hierarchy to derive specialized objects from general ones. The metamodel also incorporates module interconnection principles [9] to specify how objects relate (i.e. *connect*) to one another. Each modeling object is treated as a *module*, and a specialized form of association, called a *connection*, is used to connect one module to another. The actual connection is made between *ports* contained in the modules. If directional connections are to be modeled (as in this example) the ports can be divided into two types – *input* ports and *output* ports. Modules may contain both types of ports, depending on the type of module. For example, a `Mic` contains a single `OutputPort`, while a `PowerAmp` contains both an `InputPort` and an `OutputPort`.

In Figure 4, a `Port` is created which acts as a general module interconnection object. A `Connection` association, with `src` and `dst` roles, is used to associate (i.e. *connect*) `Port`s together. Notice that `Port`s are not used as first-class modeling objects in this paradigm, but are specialized into `OutputPort` and `InputPort` objects. `Z_Out` and `Z_In` attributes are added to the `OutputPort` and `InputPort` objects, respectively. The final modeling objects are specified as either aggregations of an `OutputPort` or an `InputPort`, or by deriving specialized versions of the `IO_Device` object which contains both an `OutputPort` and an `InputPort`. This approach represents an improvement over the metamodel of Figure 3 by more clearly defining object containment, derivation and interconnection, as well as directly associating the impedance attributes with the `OutputPort` and `InputPort` objects.

While the use of hierarchy allows a modular design approach and makes composing metamodels easier, such an approach generally requires more constraint equations to fully constrain the design. For example, the metamodel of Figure 4 allows full connectivity between *any* two `Port`-type objects (e.g. `OutputPort` or `InputPort` objects), regardless of which type of container object they appear in. This metamodel even allows an `OutputPort` to connect to itself – definitely not allowed in this paradigm. Therefore, several constraints must be placed on the metamodel.

First, the relationship between `OutputPort`s and `InputPort`s must be limited. `OutputPort`s may connect to `InputPort`s and `InputPort`s may connect to `OutputPort`s. No other connections between `OutputPort`s and `InputPort`s are allowed. The following OCL equation properly constrains this relationship:

```
Connection->forAll(c |
   c.src.oclIsTypeOf(OutputPort) and
   c.dst.oclIsTypeOf(InputPort))
```

This expression states that the `src` role of every `Connection` association must be an `OutputPort` object, and that the `dst` role must be an `InputPort` object.

The interconnections between modules must also be constrained. For example, `Mic`s can only connect to `Preamp`s, `Preamp`s can only connect to `PowerAmp`s, and `PowerAmp`s can only connect to `Speaker`s. The following set of constraints specifies these allowed connections:

```
Mic->forAll(m | m.outputPort.dst->
   forAll(i | i.preamp))

Preamp->forAll(p | p.outputPort.dst->
   forAll(i | i.powerAmp))

PowerAmp->forAll(a | a.outputPort.dst->
   forAll(i | i.speaker))
```

The first of these constraints allows `Mic`s to connect only to `Preamp`s. Note the use of `outputPort` (first letter lower case) to refer to the unnamed role at the "`Preamp` end" of the aggregation association between the `InputPort` and `Preamp` objects. Because the role is unnamed, OCL allows using the name of the associated object (beginning with a lowercase letter) as the association role name. The other two constraints function similarly to constrain connections between `Preamp`s and `PowerAmp`s, and `PowerAmp`s and `Speaker`s, respectively.

dst / src	Mic	Preamp		PowerAmp		Speaker
	Out	In	Out	In	Out	In
Mic Out	X	X	X	X	X	X
Preamp In	X	X	X	X	X	X
Preamp Out	X	X	X	X	X	X
PowerAmp In	X	X	X	X	X	X
PowerAmp Out	X	X	X	X	X	X
Speaker In	X	X	X	X	X	X

Table 2: Possible connections without constraints

dst / src	Mic	Preamp		PowerAmp		Speaker
	Out	In	Out	In	Out	In
Mic Out		X				
Preamp In						
Preamp Out				X		
PowerAmp In						
PowerAmp Out						X
Speaker In						

Table 3: Possible connections with constraints

Using OCL in this way is a powerful method for applying semantic constrains to the modeling language specification. Table 2 shows that if no constraint equations were used in the metamodel of Figure 4, 36 possible connections between the `OutputPort` and `InputPort` objects contained inside the various modules would be possible. By introducing just four constraint equations, the allowable connections are reduced to three, as shown in Table 3.

To complete this metamodel, one more constraint equation is needed to satisfy the requirement that every audio processing model have at least one `PowerAmp`. (This is the same constraint presented as part of the metamodel shown in Figure 3):

```
PowerAmp.allInstances->size >= 1
```

Before this metamodel can be used to synthesize an MGA modeling environment, the presentation specifications must be added. This is done by mapping the entities and relationships specified in the UML metamodel to various MGA-specific presentation objects. This is a fairly straightforward mapping and is not presented here to conserve space.

Conclusions and future work

This paper has presented a method for specifying MGA metamodels using UML and OCL. The method captures both abstract syntax and the static semantics of the target modeling language, and supports metamodeling composition and metamodeling tool interoperability by separating the syntactic and semantic specifications from the presentation and interpreter specifications needed to synthesize a DSME from the metamodel. An audio processing modeling paradigm was developed and used to illustrate the use of UML and OCL to state various syntactic and semantic modeling language specifications.

By using the industry standard modeling languages such as UML and OCL, potential DSME users who are familiar with these modeling languages can quickly and accurately communicate their DSME requirements to DSME developers. What remains is to apply this UML/OCL metamodeling technique to various existing and new DSME development projects to analyze the effectiveness of this approach.

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.

References

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

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

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

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

[5] Nordstrom, G., Sztipanovits, J., Karsai, G.: “Metalevel Extension of the MultiGraph Architecture”, *Proceedings of the IEEE ECBS’98 Conference*, 1998.

[6] *UML Summary*, ver. 1.0.1, Rational Software Corporation, March, 1997

[7] Quatrani, T.: *Visual Modeling with Rational Rose and UML*, Addison-Wesley, 1998.

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

[9] 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.