Formalizing the Specification of Model Integrated Program Synthesis Environments

Greg Nordstrom
Institute for Software Integrated Systems
Vanderbilt University
230 Appleton Place, Suite 201
Nashville, TN 37203
615-343-7521
greg.nordstrom@vanderbilt.edu

Abstract—Model integrated computing (MIC) is an effective and efficient method for developing, maintaining, and evolving large-scale computer-based systems (CBSs). One approach to MIC is to synthesize application programs from domain-specific models created using customized, model integrated program synthesis (MIPS) environments. The MultiGraph Architecture is a toolset for creating graphical domain-specific MIPS environments (DSMEs). By modeling the syntactic, semantic, and presentation requirements of a DSME, a *metamodel* is formed and 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 paper presents a method to represent DSME requirements using UML class diagrams and predicate logic constraint language expressions, and discusses automatic transformation of metamodel specifications into DSMEs.

TABLE OF CONTENTS

1. 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 environments include large-scale production facility process monitoring; real-time diagnostics and analysis of manufacturing execution systems; web-based information distribution, integration, 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 importance 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 system'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 or input data streams for simulation and/or analysis tools.

One approach to MIC is model-integrated program synthesis (MIPS). In MIPS, formalized models capture various aspects of a domain-specific system's desired structure and behavior. Model interpreters are used to perform the computational 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, the MultiGraph kernel [2], POSIX, etc.), or to supply input data streams for use by 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 generating 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? In all likelihood, the existing DSME would not be able to model new configurations of the plant. Instead, the entire DSME must be upgraded to allow models of the new equipment to be incorporated into existing and future chemical plant models.

The MGA tools have been used to develop MIC solutions for computer-based systems for over 10 years [3] [4] [5] [6] [7] [8]. 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* – 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.

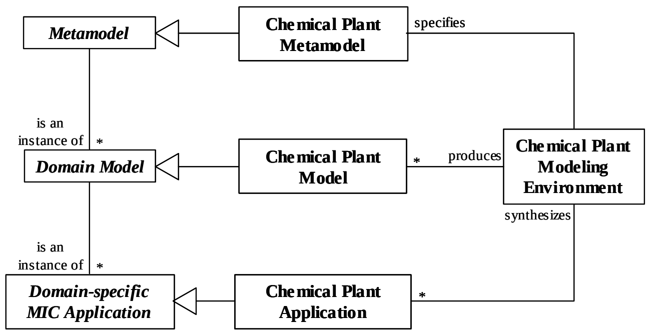


Figure 1 Modeling and metamodeling relationships

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, and discusses automatic transformation of such metamodels into DSMEs.

2. MODEL-INTEGRATED PROGRAM SYNTHESIS

Modeling reduces design cycle times, allows completeness and consistency checking throughout the design process, aids in documenting the design itself, 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 [9]. 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.

A MIPS environment operates according to a domain-specific *modeling paradigm* – a set of requirements that govern 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 aspects necessary to logically and efficiently partition the design space; how semantic information is to be represented in, and later extracted from, the models; 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.

3. 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, metaobject, etc.) Unfortunately, the prefix 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 a 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 and is in general use by the metamodeling community. The following table, taken from [10], 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 [10]. The previous definitions for Model, Metamodel, and Meta-metamodel correspond to the upper three layers of Table 1.

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 [10], 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 the case of a multi-aspect graphical modeling language, where models are to be viewed from different aspects or points of view, the metamodel must clearly define a partitioning of the graphical constructs into such aspects.

Modeling 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 in the context of model instances themselves. Only static semantics may be specified in a metamodel – the metamodel has no way of knowing *a priori* what meaning to associate with particular instances (i.e. particular models) created using the language.

Another consideration in any metamodeling language is the form of these invariant constraint statements. Constraints should be analyzable, allowing automated or semi-automated consistency checking before synthesizing a modeling environment. This requires that they 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, the presentation specifications must be defined. Part of this specification is the mapping of graphical modeling idioms available in the target modeling 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 of the modeling language. Therefore, metamodeling tools and environments must be able to accept metamodels specified in a variety of metalanguages, or those 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.

Metamodel Composition and Translation

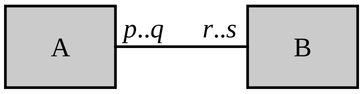
Because common modeling concepts apply to a wide variety of engineering domains, the approach to creating DSMEs is to customize (i.e. configure) a general graphical modeling environment for use in a particular domain according to specifications included in a metamodel. This is done by representing general 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 principles

Name	Description
Module Interconnect	Provides rules for connecting objects together and defining interfaces. Used to describe relationships among objects.
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 principles. These principles represents constraints on the modeling process. Because of their general nature, the principles must be customized before being used in a given metamodel. This allows the metamodeler to inject domain-specific concepts into the metamodel. One approach to this customization is parameterization.

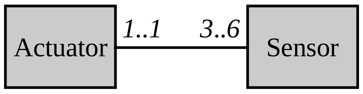
Figure 2 defines a general constraint on binary object association (shown both graphically and textually). This constraint specifies that an object of type A can be associated with between r and s objects, inclusive, of type B, and that objects of type B can be associated with between p and q objects, 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 a DSME for modeling aircraft in-flight safety systems, where between three and six engine temperature sensors can be associated with a single fire suppression system actuator. The general object association constraint from Figure 2 is parameterized for this particular domain as follows:



```
DomainConstraint:ObjAssociation();
   .
   .
   .
DomainConstraint(Actuator,1,1,Sensor,3,6)=true;
```

Figure 3 Customized 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 principles in this manner represents specifying the DSME's modeling language syntax. 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, since they only apply to the particular domain.

Figure 4 shows how metamodels are composed by tailoring general model composition constraints and combining them with constraints specific to the domain.

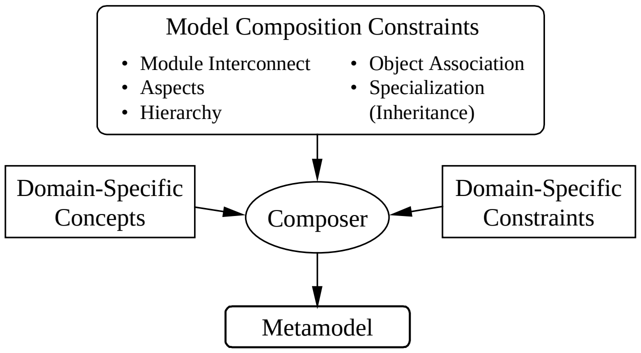


Figure 4 Metamodel composition

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 DSME 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 [11] for a discussion of the theory and practice of specifying interpreter behavior.

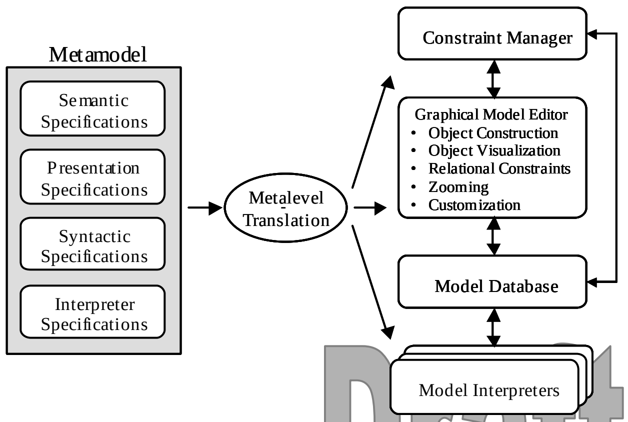


Figure 5 Metamodel translation

As mentioned earlier, DSME synthesis and evolution is done by translating the metamodel to configure the elements of a general modeling environment, creating the DSME. Figure 5 shows a metamodel containing the necessary specifications. Translating these specifications into a form suitable for configuring the DSME is done by the metalevel translator. The Constraint Manager, which is responsible for ensuring that only valid models are created in the target domain, is configured using information from 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, and how to control the type and multiplicity of object associations. Model Interpreters are partially configured using information from the metamodel's interpreter specification.

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 paradigm, and as such, appear as domain-specific modeling constraints in the metamodel. Such constraints can only be enforced in the presence of actual domain-specific models – model instances created using the modeling language specified by the metamodel. Enforcing these constraints is done by the constraint manager. The constraint manager is part of the domain-specific modeling environment. It provides various queues to the modeler according to the static semantics described in the metamodel.

Consider again the processors and sensors example. It was stated earlier that sensors can be connected to processors, and vice versa. By specifying the entities (processors and sensors), the connection roles and multiplicities, and a simple mapping to particular graphical objects, the resulting sensor-to-processor connection specification could be easily enforced by a graphical model editor – the graphical editor would only allow interconnections between processors and sensors, and disallow all other types of connections (e.g. processor-to-processor, 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, by including an invariant expression stating that the size of the set of processor 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. Of course, the constraint manager can't guarantee certain editing actions either, but it can indicate that, at a given point in time, a certain model does not satisfy a particular constraint.

4. Using the Metamodeling System

To illustrate the process of specifying and synthesizing a DSME, a simple audio processing system modeling environment will be created. The DSME requirements are:

- Audio systems are to be modeled using microphone, preamp, power amp, integrated amp, and speaker components.
- Integrated amps contain one or two preamps and one or two power amps, along with input and output ports. Connections between input ports, preamps, power amps, and output ports indicate signal flow paths within an integrated amp.
- Audio systems consist of at least one microphone, one integrated amp, and one speaker. Connections between microphones, the input ports and output ports of integrated amps, and speakers indicate signal flow paths in the audio system.
- Microphones connect to the input ports of integrated amps. The output port of every integrated amp must be connected to at least one speaker.
- Modelers must be able to create models of both mono and stereo audio processing systems.

The Audio Processing System Metamodel

Figure 6 below shows the UML portion of the audio processing system metamodel. This specification is a direct representation of the requirements listed above.

Figure 6 specifies the types of modeling objects allowed (e.g. `Mics`, `Preamps`, etc.), object attributes (e.g. `Rating`), and association types allowed among the objects (e.g. `PreToPower`). The metamodel shows that `IntegratedAmp`s are made up of one or two `Preamps`, one or two `PowerAmps`, and zero or more `InPort`s and `OutPort`s. A `System` consists of one or more `IntegratedAmp`s along with one or more `Speaker`s and/or `Mics`. A UML `Note` is used to hold metamodel version information. Within an `IntegratedAmp`, `InPort`s connect to `PreAmps`, `PreAmp`s connect to `PowerAmps`, and `PowerAmps` connect to `OutPort`s. `Mics` connect to the `InPort`s of `IntegratedAmp`s and

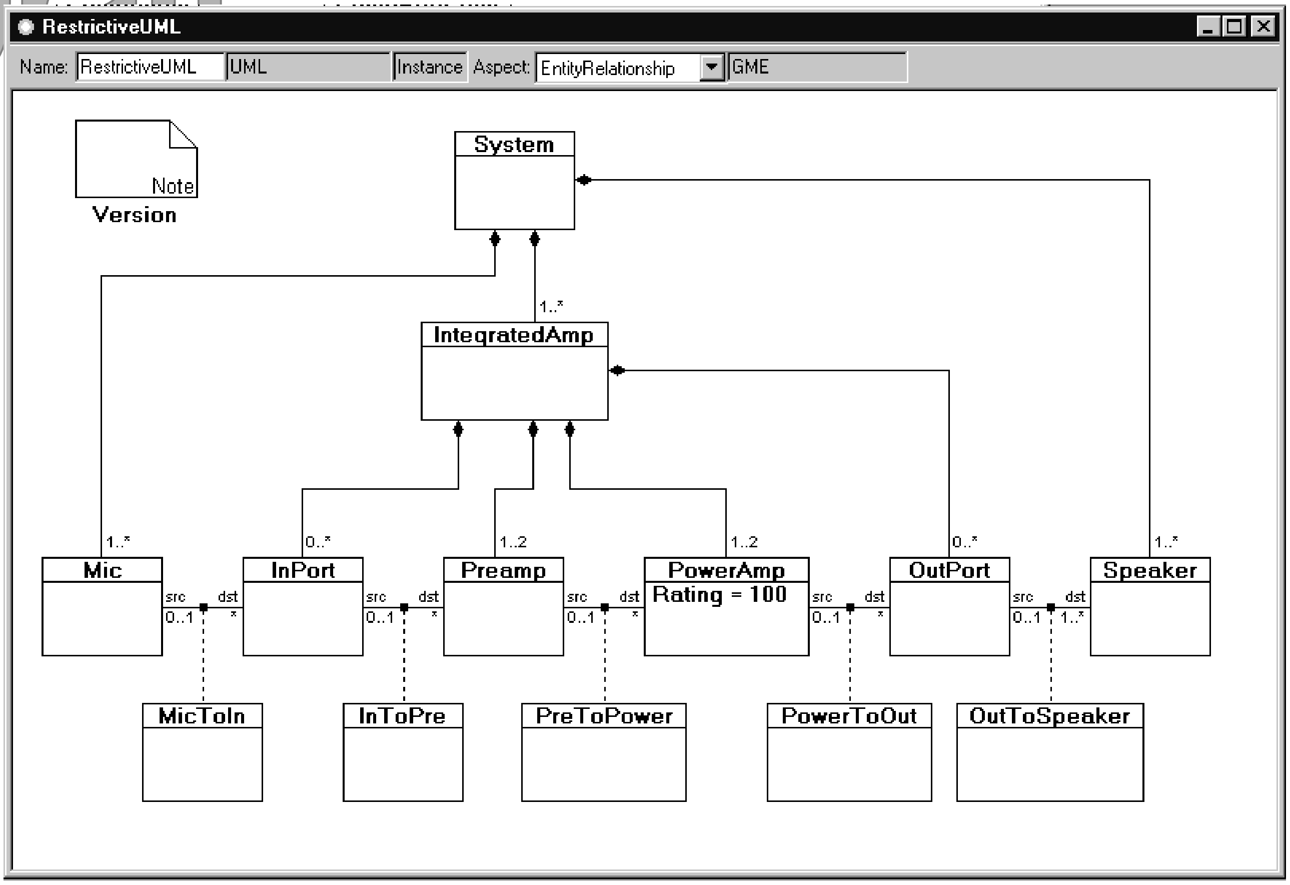


Figure 6 UML portion of the audio processing system metamodel (initial version)

`OutPort`s of `IntegratedAmp`s connect to `Speaker`s, forming models of audio processing systems (interconnections show signal flow paths).

While Figure 6 captures the audio processing system requirements as listed above, it represents a fairly restrictive approach to specifying a DSME. Classification of similar component types is not used, and many associations between objects need to be defined (e.g. `MicToIn`, `InToPre`, `PreToPower`, etc.). Such a brittle DSME design will generally require significant modification as domain requirements change. For example, if a new signal processing component such as a noise gate were added to the domain and allowed to connect between `Mic`s and the `InPort`s of `IntegratedAmp`s, a noise gate object would need to be added to the metamodel, the existing `MicToPre` connection would have to be discarded, and two new connection specifications (e.g. `MicToGate` and `GateToIn`) would have to be created between the noise gate and the `Mic` and `InPort` objects.

Figure 7 shows a more general approach. Here, abstract `Component` and `Port` objects are defined. As before, `IntegratedAmp`s contain `Port`s, but a more general `CompPortConn` association is used to allow specialized `Mic`, `Preamp`, `PowerAmp`, and `Speaker` components to connect to port-type objects (e.g. `InPort`s and `OutPort`s) contained within `IntegratedAmp` components. Such a module interconnection design approach [12] (i.e. connecting modules together via encapsulated I/O ports) allows easier evolution of the modeling environment as domain requirements change over time. Adding a noise gate to the metamodel of Figure 7 involves deriving the new noise gate object from the `Component` object and aggregating it into the `System` object. No new connection specifications would be required, since the "connectivity" of component objects is not changed when new components are added.

Of course, the generalized approach of Figure 7 represents a trade-off over the more specific design of Figure 6. Consider the previously stated requirement that the output of every `IntegratedAmp` be connected to at least one `Speaker`. Such a requirement was easily modeled directly in Figure 6, but cannot be modeled using the UML diagram of Figure 7 due to the general nature of `Port` objects. A *constraint* on the `OutPort` objects contained within `IntegratedAmp` objects must be specified.

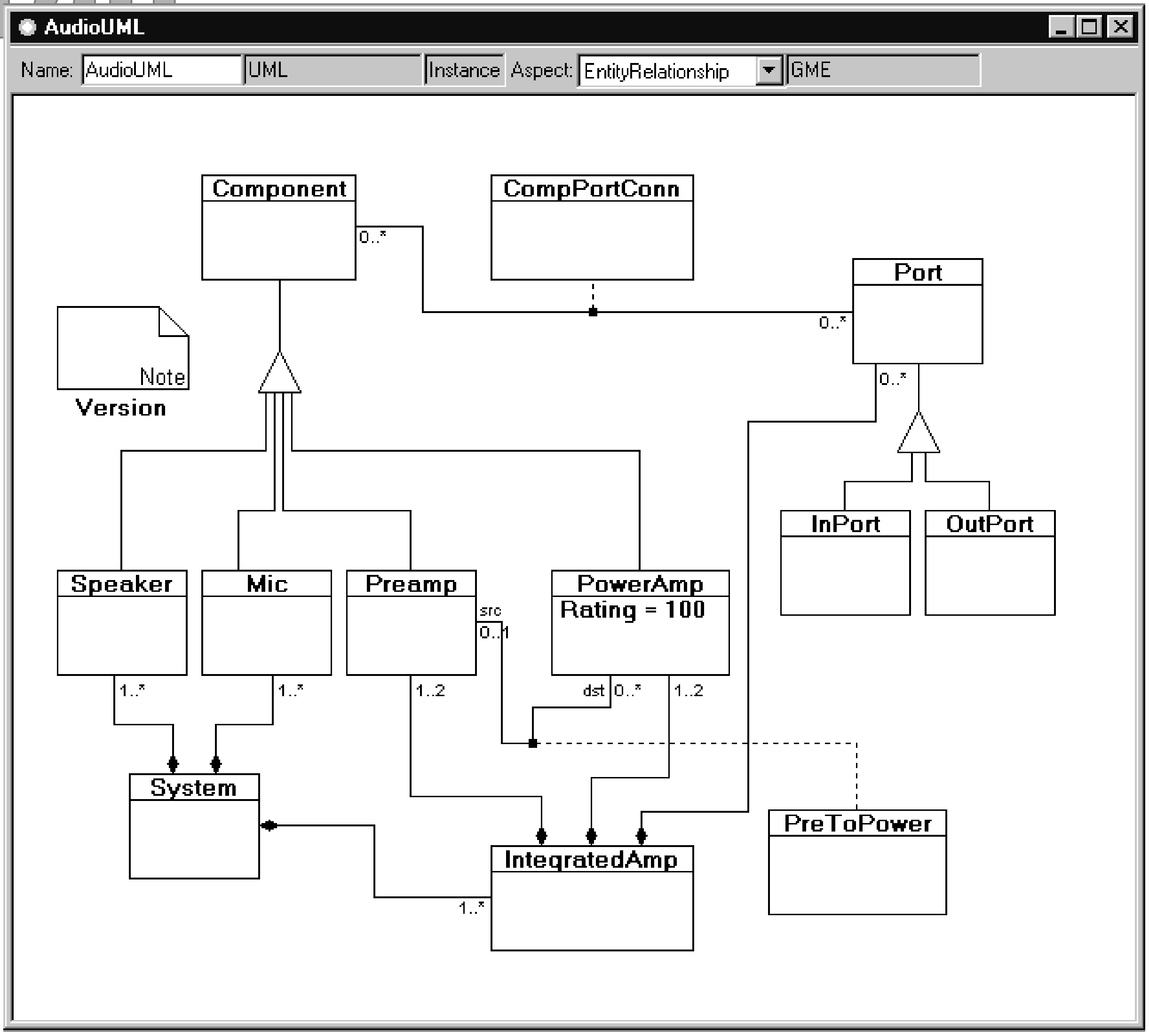


Figure 7 UML portion of the audio processing system metamodel (final version)

Constraints in the MGA metamodeling environment are expressed using the MGA Constraint Language (MCL), a predicate logic language based on OCL [13] [14]. MCL uses a syntax and a semantics similar to OCL, but also includes expressions for collections of specific kinds of MGA objects such as `models` (MGA container objects) and `parts` (contained objects). An MCL constraint specifying that every `IntegratedAmp` be connected to at least one `Speaker` is listed below.

```
models(“IntegratedAmp”)->
  forAll(m|m.parts(“OutPort”)->
    connectedTo(“Speaker”)->size() > 0)
```

This invariant expression states that the size of the set of `Speaker` objects that each `OutPort` contained within an `IntegratedAmp` connects to must be greater than zero. (Here, `models(“IntegratedAmp”)->` returns the set of all `IntegratedAmp` models in a given audio system, and `m.parts(“OutPort”)->` returns the set of `OutPort` objects contained within each `IntegratedAmp`). In other words, the output of every `IntegratedAmp` must be connected to at least one `Speaker`. *NB*: The `CompPortConn` association specifying that connections may exist between `Component`- and `Port`-type objects is made at a hierarchically abstract level within the design – between the abstract classes `Component` and `Port`. While such abstract associations allow for modular designs and make designing and composing metamodels easier, such an approach generally requires more constraint equations than designs with no abstract objects.

The metamodel is not yet complete. What remains is to establish a presentation specification that maps the general modeling concepts expressed as UML objects and associations onto the available MGA modeling resources

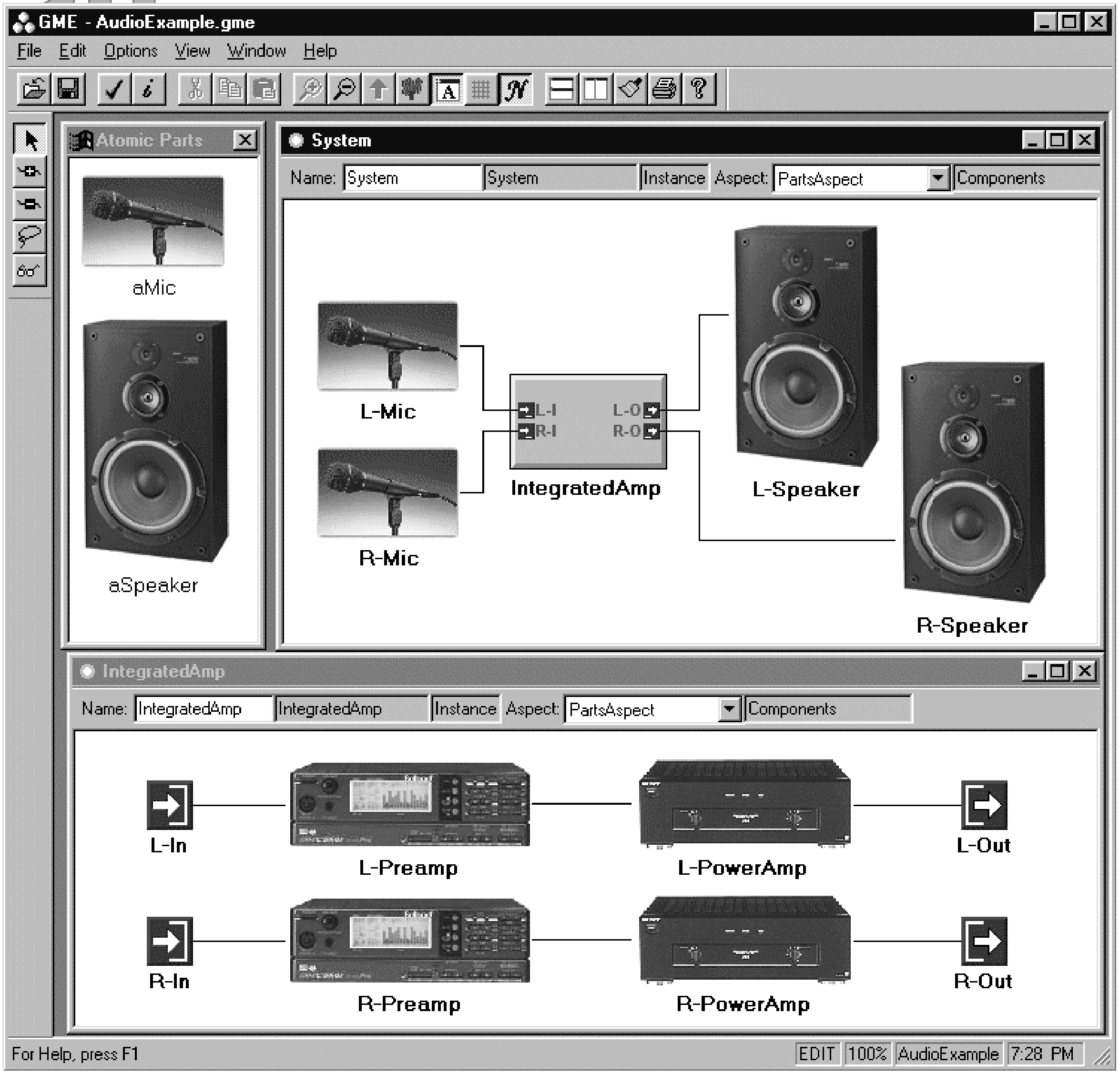

GME - AudioExample.gme
File Edit Options View Window Help
Atomic Parts
aMic
aSpeaker
System
Name: System System Instance Aspect: PartsAspect Components
L-Mic
R-Mic
L-I R-I L-O R-O
IntegratedAmp
L-Speaker
R-Speaker
IntegratedAmp
Name: IntegratedAmp IntegratedAmp Instance Aspect: PartsAspect Components
L-In
L-Preamp
L-PowerAmp
L-Out
R-In
R-Preamp
R-PowerAmp
R-Out
For Help, press F1
EDIT 100% AudioExample 7:28 PM

Figure 8 Synthesized audio modeling environment

(i.e. the MGA graphical modeling objects and relationship mechanisms). Table 3 shows which MGA resources can be used to represent various general modeling concepts. MGA resources are discussed in detail in [15].

Table 3 Representing general modeling concepts using MGA modeling resources

General Modeling Concept	**MGA Modeling Resource**
Module Interconnect	Models containing Atomic Parts (playing the role of interconnection ports)
Multi-Aspect Modeling	Aspects
Hierarchy	1. Model/Atomic Part containment 2. Conditionalization
Object Association	1. Binary Connections 2. Atomic Part- and/or Model References; References to References
Specialization (a.k.a. Inheritance)	N/A (possibly using references, attributes)

The details of the presentation specification mapping are beyond the scope of this paper. However, once the mapping has been defined, the metamodel is complete, and can be used to generate the DSME itself. Figure 8 below shows the resultant audio processing system modeling environment being used to model a simple stereo audio system. The lower MGA model is a detailed view of the `IntegratedAmp` component shown in the center of the upper (“`System`”) MGA model.

5. Conclusions

This paper has presented a method for formally representing DSME requirements as metamodels using UML class diagrams and predicate logic constraint language expressions which can be used to synthesize a target DSME. The DSME can then be used to create models of domain-specific systems. Model interpreters are used to apply semantic meaning to the models, forming a basis for model analysis and for translation of the domain models into executable models or data streams required by third-party analysis and/or execution environments.

Such a metamodeling approach has three distinct advantages. First, by formalizing the DSME specification process, DSME design requirements can be stated hierarchically and refined using information as it becomes available. The formalized specifications also 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 requirements change.

Current and future research in DSME specification and generation focuses on two goals: making better use of existing metamodel design formalisms (e.g. UML and MCL), and allowing interpreter specifications to be stated formally in metamodels. It is anticipated that libraries of metamodel solutions to "standard" modeling problems will be developed and incorporated into the metamodeling environment. 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.

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] 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] E. Long, A. Misra, J. Sztipanovits: "Increasing Productivity at Saturn", *IEEE Computer Magazine*, August, 1998.

[4] G. Karsai, F. DeCaria: "Model-Integrated On-line Problem-Solving Environment for Chemical Engineering", *IFAC Control Engineering Practice*, Vol. 5, No. 5, pp. 1-9, 1997.

[5] G. Karsai, S. Padalkar, H. Franke, Sztipanovits J.: "A Practical Method For Creating Plant Diagnostics Applications", *Integrated Computer-Aided Engineering*, Vol. 3, No. 4, pp. 291-304, 1996.

[6] S. Padalkar, G. Karsai, J. Sztipanovits, F. DeCaria: "Online Diagnostics Makes Manufacturing More Robust (Part 1)", *Chemical Engineering Magzine*, pp. 80-83, 1995.

[7] G. Karsai, J. Sztipanovits, S. Padalkar, C. Biegl: "Model Based Intelligent Process Control for Cogenerator Plants", *Journal of Parallel and Distributed Systems*, pp. 90-103, 1992.

[8] J. Sztipanovits, J. R. Bourne: "Architecture of Intelligent Medical Instruments", *Journal of Biomedical Measurements Informatics and Control, London, UK.*, Vol. 1, No. 3, pp. 140-146, 1987.

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

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

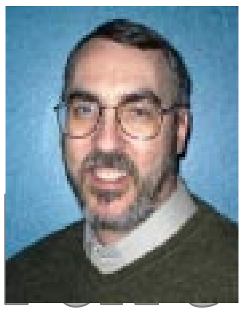
[11] G. Karsai, et al., "Towards Specification of Program Synthesis in Model-Integrated Computing," Proceedings of the IEEE ECBS'98 Conference, 1998.

[12] 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.

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

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

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



***Greg Nordstrom** is an Associate Research Professor at Vanderbilt University's Institute for Software Integrated Systems where much of his work is in the area of graphical modeling language specification. He received the B.S. degree in electrical engineering from Arizona State*

University in 1987, the M.S. degree in electrical and computer engineering from the University of Tennessee Space Institute in 1992, and the Ph.D. in electrical and computer engineering from Vanderbilt University in 1999.