Meta-level Extension of the Multigraph Architecture

Greg Nordstrom, Janos Sztipanovits and Gabor Karsai
Measurement and Computing Systems Laboratory
Vanderbilt University

Abstract

Domain-specific model integrated program synthesis (MIPS) environments are created according to a modeling paradigm – a description of the class of models that can be created using the system. Just as model integrated computing applications are executable instances of domain models, domain models can be viewed as instances of meta-models. Desired modeling environment characteristics are selected from a library of model composition constraints and combined with domain-specific modeling concepts and constraints to create a meta model. This meta model is then translated and used to automatically generate the actual MIPS environment. Advantages include support for formal specifications of domain-specific modeling paradigms and model interpreters, fast adaptation of applications through the automatic re-synthesis of running applications from models, evolution of applications through model modification, and slow evolution through incremental modification of the MIPS environment components.

Background

Large computer-based systems operate in ever-changing environments. Throughout a system's life cycle, changes in mission requirements, personnel, hardware, support systems, etc., all drive changes to the system. Rapid reconfiguration via software has long been seen as a potential means to effect rapid change in such systems. However, due to the extremely 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 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 a method to state the system's requirements and design in concise and unambiguous terms, the ability to automatically transform requirements and design into software, and a method to safely and efficiently modify the software throughout its lifetime. One promising technology to aid in such evolution is model integrated computing.

Model-integrated computing

Model-integrated computing (MIC) environments synthesize software from domain-specific models of the desired system. Multiple models are used to capture the behavior of the system's software, its environment (e.g. system hardware, external data flows, user interfaces, etc.), and the interaction between the two. Thus, model integrated computing is well suited for systems where there is a high degree of coupling between the software and its environment.

The MIC environment can be broken down into three main layers of abstraction. The top, or meta-level, layer contains the metaprogramming interface and meta-level translators. Here, formal descriptions of particular modeling environments are created using formal specification languages. Examples include modeling environments for process control and monitoring; real-time diagnostics and analysis; information distribution and management; surety of high consequence, high assurance systems; and fault detection, isolation, and recovery.

Once the meta-level specifications have been created and validated, they are translated for use by the next layer, the model integrated program synthesis (MIPS) layer. The MIPS layer uses information from the meta-level translators to synthesize domain-specific modeling environments. These domain-specific modeling environments are then used to create and analyze domain-specific models of individual systems. Model interpreters are used to synthesize the actual executable applications.

The Multigraph Architecture

The Multigraph Architecture (MGA), currently under development at Vanderbilt University's Measurement and Computing Systems Laboratory, is a model integrated computing infrastructure. The MGA provides a layered software architecture and framework for building domain-specific environments. Such environments are capable of: (1) constructing, testing, and storing domain-specific models, (2) transforming these domain models into both analyzable and executable models, and (3) integrating applications on heterogeneous computing platforms.

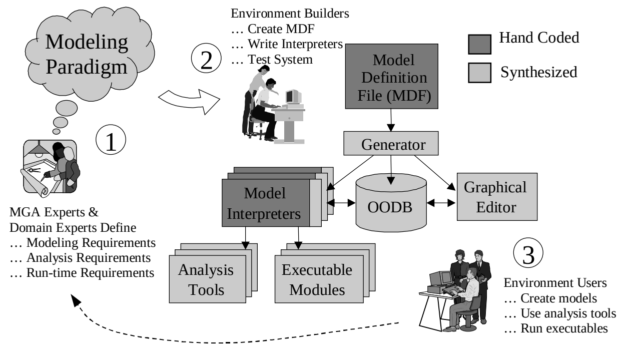

Figure 1. Creating a modeling environment using the Multigraph Architecture (MGA)

Figure 1 shows how the MGA is used to create domain-specific modeling environments. The process begins by formulating the domain's *modeling paradigm*. The modeling paradigm contains all the syntactic and semantic information regarding the domain – which concepts will be used to construct models, what relationships may exist among those concepts, how the concepts may be organized and viewed by the modeler, and rules governing the construction of models. The modeling paradigm defines the *family* of models that can be created using the resultant MIPS environment.

Both domain- and MGA experts participate in the task of formulating the modeling paradigm. Experience has shown that the modeling paradigm changes rapidly during early stages of development, becoming stable only after a significant amount of testing and use. A contributing factor to this phenomenon is the fact that domain experts are often unable to initially specify exactly how the modeling environment should behave. Of course, as the system matures, the modeling paradigm becomes stable. However, because the system itself must evolve, the modeling paradigm must change to reflect this evolution. Changes to the paradigm result in new modeling environments, and new modeling environments require new models. Model migration is very much an open issue.

Once a paradigm is decided upon, the environment builder creates a model description file (MDF). The MDF contains a formal, declarative language description of the paradigm's model construction semantics – what types of objects may be used to construct models, how those objects will appear on-screen, and how they may be associated (i.e. connected) with each other. The MDF also contains descriptions of any hierarchical and/or multi-aspect model creation and viewing properties that must exist in the modeling environment

Once created, the MDF is used to automatically generate the graphical model editor, the object database schema, and database interface code. At this point, the domain-specific modeling environment can be used to create models of specific systems within the domain. However, because the MDF specifies only the model construction semantics of the modeling paradigm, no run-time meaning can be inferred from the models. Making sense of the models, i.e. *interpreting* the models, is the job of the model interpreters, which access and process data from the populated object database for use by the analysis tools and executable modules.

Although the MDF and the model interpreter code actually form an informal, de facto specification for the MIPS environment, there is no single specification that could be used to formally validate the consistency of the modeling concepts represented by the MDF and the model interpreters. Paradigm validation is necessary to ensure that the environment sufficiently constrains the modeler, so that only legal models can be created. Only a careful examination of the MDF and considerable amounts of testing can uncover such inconsistencies – a time consuming and error prone process for humans.

Meta-modeling

To address the current shortcomings of the MGA, meta-modeling techniques are being developed. Meta-modeling offers several advantages. First, standardization is achieved by using a single MIPS environment specification – a *meta model*. The meta model is written using a formal specification language, and can be constructed by combining predefined general modeling constraints with domain-specific constraints and concepts.

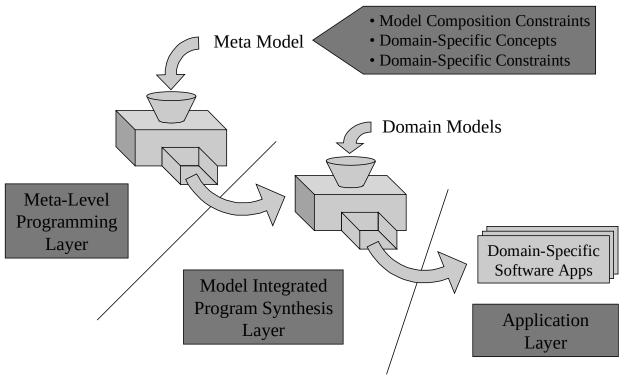
Meta Model
• Model Composition Constraints
• Domain-Specific Concepts
• Domain-Specific Constraints
Domain Models
Meta-Level Programming Layer
Model Integrated Program Synthesis Layer
Domain-Specific Software Apps
Application Layer

Figure 2. MIPS environment and domain-specific application synthesis using the MGA

Such a modular approach makes constructing the meta model easier, and contributes to a better understanding of the paradigm – a critical factor when initially creating a MIPS environment. In this way, less iteration is required to settle on a "good" modeling paradigm. Also, because the meta model is written using a formal specification language, it can be reasoned about with the help of a proof checker to uncover inconsistencies in the paradigm before a MIPS environment is created, allowing for faster, easier, and safer system development and evolution.

The MGA has been extended to include a meta-programming level as shown in Figure 2. Meta models, consisting of general engineering model composition constraints as well as domain-specific concepts and constraints, are used to synthesize domain-specific MIPS environments. These domain-specific MIPS environments are then used to create domain-specific models and to synthesize domain-specific applications based on those models.

Composing meta models

The field of engineering contains many domains, such as process control, digital signal processing, and information distribution and management to name a few. Each domain deals with inherently different notions, but because they are all engineering systems, certain modeling concepts can be applied to them as a whole. For example, hierarchy is often used as a means to represent information hiding, and module interconnection principles are used to specify information flow. Such principles represent *constraints* when applied to a particular domain. Because such modeling constraints can be reused across engineering domains, meta-models can be constructed quickly and accurately from a standardized "library" of engineering modeling constraints. This assumes, of course, that each constraint has been validated beforehand, and that combining constraints does not invalidate such compositions or that re-validation of the composed specification is possible.

Referring again to Figure 2, MGA meta models are formed by composing general model composition constraints with domain-specific concepts and constraints. Meta-level translators are then used to automatically create the necessary configuration files needed by the program synthesis environment.

Applying domain-specific concepts involves selecting certain model composition constraints for inclusion in the meta model. Not all domains need every constraint available in the library. For example, a simple process control modeling application may not require hierarchical behavior. For this reason, when creating domain models, concepts specific to the domain must be known *a priori* and must be used to guide the selection of constraints from the library.

Constraint Name	Constraint Description
Module Interconnect	Provides rules for connecting objects together. 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 1. Model composition constraints

Table 1 describes these model composition constraints. Note that these are not specific to a single domain, but are written in such a way as to allow application to many engineering domains. Because library constraints have been predefined and pre-tested, meta models can be constructed quickly and easily by selecting specification modules according to the particular domain-specific concepts required for the resulting MIPS environment. While this modular construction method is quite useful, it is not enough to adequately describe a particular modeling domain. The constraints listed in Table 1 are too general. They are "family-specific" (e.g. engineering-specific) but not domain-specific. Before a domain specification is complete, these model composition constraints must be customized for the specific domain. This is done by adding domain-specific constraints.

Composition occurs via several mechanisms. The simplest method is to include one specification in another. This allows the meta models to be built up in a modular fashion. When included specifications are too general for a particular application, the modeler may choose to translate the specification into a more specific form. For example, the following partial specification, written in Specware [3], describes a simple binary relation.

```
spec BINARY-RELATION is
   sorts Domain, Range
   op related? : Domain, Range -> Boolean
end-spec
```

This specification says that binary relations consist of domains and ranges. The Boolean operation `related?` allows the specifier to assert that a certain domain is related to a certain range. Such a specification is much too general for use in a meta model. However, by importing and translating the sorts and operation contained in this specification, a more domain-specific specification, called `CONNECTION`, can be created as shown below.

```
spec CONNECTION is
    translate BINARY-RELATION by
    { Domain   -> Source,
      Range    -> Destination,
      related? -> connected? }
end-spec
```

The resultant spec, `CONNECTION`, can now be used to specify connections, which are made up of a source connected to a destination. The binary operation `connected?` can be used to assert that a particular connection exists.

Another method for composing specifications with Specware is the *colimit*. The colimit is used to create a new specification from the union of two or more existing specifications. The specifier indicates which portions of each specification are to be shared in the union. For example, one specification, named `SIZE`, may contain axioms constraining an object's size, while another specification, `COLOR`, may contain axioms constraining an object's color. A new specification, `IMAGE`, could be created from the colimit of `SIZE` and `COLOR` which could be used to constrain both an object's size *and* color, as shown below.

```
spec SIZE is
    axiom (height 10)
    axiom (width 5)
end-spec
spec COLOR is
    axiom (color “blue”)
end-spec
spec IMAGE is
    colimit of diagram
        nodes
            E : TRIV,
            SIZE,
            COLOR
        arcs
            E -> SIZE
            E -> COLOR
        end-diagram
end-spec
```

The reader should not be concerned with the specifics of Specware at this point, but recognize that this mechanism allows axioms from two separate specifications to be included in a newly created `IMAGE` specification. In this example, `IMAGE` specifies objects that are of height 10, width 5, and blue in color. The node `E:TRIV` is used to “glue” together the specifications `SIZE` and `COLOR`.

Proof-of-concept example

To assess the feasibility of applying meta-modeling techniques to the MGA, a simple MIPS environment was synthesized from a meta model. The example demonstrates several key meta-modeling concepts, such as formal specification of modeling constraints, specification composition, and automatic synthesis of an MDF.

Two model composition constraints (refer to Table 1. above) included in this example are module interconnection and object association. The example also incorporates two domain-specific constraints – constrained binary relationships and multiplicity.

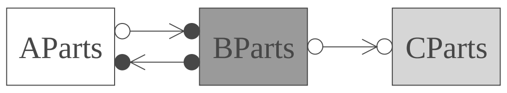

Figure 3. Graphical meta model

Figure 3 is an Object Modeling Technique (OMT) [1] object diagram which represents the modeling domain used in this example. This object diagram is a meta model which describes the relationships which are allowed to exist in our modeling domain. The lines indicate which objects can be connected together, the arrowheads[1] indicate which object is the destination when connecting a pair of objects, and the circles indicate how many of each object can be connected together. Hollow circles indicate zero or one object, while darkened circles indicate zero to many. For example, when connecting AParts source objects to BParts destination objects, the AParts object can be connected to at most one BParts object.

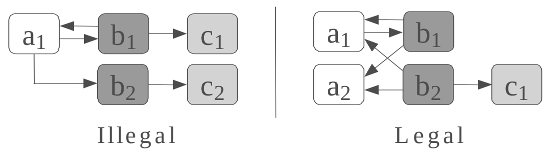

Figure 4. Two domain models

Two possible domain models for our example environment are shown in Figure 4. These are both instances of the meta model in Figure 3. However, the model on the left is illegal, since it violates the connection rule established in the meta model for connecting AParts source objects to BParts destination objects.

[1] OMT uses a numbering scheme to indicate the source and destination when connecting objects together. We have substituted the arrowheads for clarity in this example only.

Formal specification language

A model is said to be *valid* (or *legal*) if it conforms to a given modeling paradigm. A model is *correct* if it faithfully represents reality. Assuming that the modeling paradigm is correct, we can state that all correct models are valid models. However, the converse may not be true – valid models are not necessarily correct models. For example, imagine that an existing power plant is being modeled, but the modeler fails to include a critical plant component, such as an over-current detector. In this case, the model is valid (the modeling paradigm does not *require* over-current detectors, but merely *allows* them), but incorrect – it does not represent reality. However, if the modeling paradigm required every model to include the over-current detector, the model would be invalid.

Why draw such a distinction? Looking again at Figure 4, how does one become convinced that the model on the left is, in fact, an illegal model? Only by *reasoning* about a particular domain model in light of the meta model can the domain model's validity be determined. And while reasoning in this manner may work for small meta models, as the size and complexity of the modeling paradigm grows, the meta modeler cannot be expected to validate a meta model by mental reasoning alone. If, however, the meta model is expressed in a formal specification language, the meta modeler can use a computer to aid in the task of ensuring meta model consistency[2].

The next question becomes "which formal language to use?" The answer depends on the goals of the meta modeling activity. In the case of the MGA, there are four main goals driving the move toward meta modeling.

First, the formal language must be able to represent the types of constraints that the meta modeler expects to encounter. In the case of OMT, the modeler is faced with a finite set of possibilities. Only so much can be represented using OMT, and no facility exists for the modeler to extend OMT to cover new situations. Thus, graphical modeling "standards" such as OMT or UML [2] are not adequate for describing the modeling relationships found in typical MGA applications. A better choice in this case is one of the many programmable formal specification languages, such as Specware [3][4], Larch [5], or PVS [6].

Second, meta models must be able to be assembled from general, pre-existing specifications of model composition constraints (refer to Table 1. above for descriptions of the constraints.) The meta modeler must be able to easily combine specifications to form a meta model that accurately reflects the requirements represented in the modeling paradigm. Therefore, the formal language must have mechanisms for combining one specification with another, and for customizing or refining the specification from a general form to a more domain-specific form. Specware was used earlier to show how this can be accomplished, but Specware is not unique in this respect. Other languages also have the capability to create and refine specifications from existing ones.

Third, because the meta modeler will use computers to aid in checking model consistency, the formal language must support a theorem prover. Once the modeler believes the meta model contains the proper axioms describing the modeling paradigm, theorems must be developed by which to test the meta model. Only then will the meta modeler have sufficient confidence in the meta model to use it to create a MIPS environment.

Finally, because the MIPS environment will be synthesized from the meta model, the meta modeler must be able to transform the meta model into a form suitable for use by the MGA. Therefore, the formal language must be an open language, supporting access to its internal data structures, so that key pieces of information may be extracted and used in the transformation process.

Object association and multiplicity

Figure 3 above describes two important requirements of our example interconnection environment – the number (multiplicity) and type (object association) of objects that can be connected together. To develop these requirements into formal specifications it is necessary to examine the mathematical foundation of object association and multiplicity. This foundation is rooted in set theory and first order logic. The following discussion is based on work done by Bourdeau and Cheng [7].

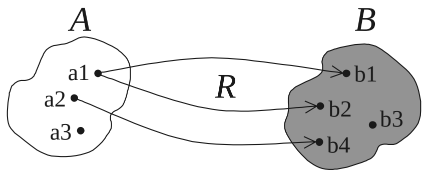

Figure 5. Set theory representation of an injective relationship

Figure 5 is an example of a relationship *R* between *A*- and *B*-type objects. The diagram shows that every element of *B* is related to at most one element of *B*. Such a relationship is called an *injective binary relationship between A and B*. The relationship can be written as a relational predicate as follows.

$$\forall\ x, y{:}A, b{:}B\ .\ (R(x, b) \wedge R(y, b) \Rightarrow x=y)$$

[2] Consistent meta-models will lead to modeling environments which are *better able* to prevent a system modeler from building invalid, if not incorrect, models. Inconsistent meta-models will almost certainly lead to incorrect models!

This equation reads that for all *x* and *y* of type *A* and all *b* of type *B*, if *x* is related to *b* and *y* is related to *b* then *x* must equal *y*.

Functional (*R*,*A*,*B*)	Every element of *A* is related to *at most* one element of *B*
Injective (*R*,*A*,*B*)	Every element of *B* is related to *at most* one element of *A*
Surjective (*R*,*A*,*B*)	Every element of *B* is related to *some* element of *A*
Total (*R*,*A*,*B*)	Every element of *A* is related to *some* element of *B*

Table 2. Basic relationships used to develop multiplicity constraints

Table 2. describes the four key relationships needed to formally describe binary object associations with multiplicity. By combining these relationships, any binary object association with multiplicity can be described.

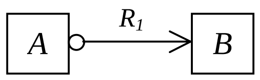

Figure 6. A zero-to-one to one binary relationship

Figure 6 shows an OMT relationship that allows zero or one *A*-type objects to be associated with exactly one *B*-type object. Such a relationship can be described by the conjunctive predicate formula $R_1(A, B)$ = injective(A, B) $\wedge$ total(A, B) $\wedge$ functional(A, B).

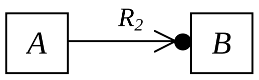

Figure 7. A one to zero-to-many binary relationship

Similarly, Figure 7 shows a relationship that allows exactly one *A*-type object to be associated with zero or more *B*-type objects. This relationship can be written as $R_2(A, B)$ = surjective(A, B) $\wedge$ injective(A, B).

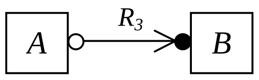

Figure 8. A zero-or-one to zero-to-many binary relationship

By combining OMT diagrams, new diagrams can be obtained. Figure 8 shows the result of combining Figures 9 and 10 to obtain a zero-to-one to zero-to-many relationship. Mathematically, this relationship becomes $R_3(A, B) = R_1(A, B) \cap R_2(A, B)$ = injective(A, B). Thus, new relationships can be formed by taking the *intersection* of existing relationships. This important concept allows complex specifications to be composed from existing, more general specifications, as discussed in the previous section.

Specification code

The following code fragments demonstrate the key meta-modeling concepts contained in this example. A complete listing of Specware code is available from the authors upon request. The code begins with a simple specification describing a binary connection.

```
spec BINARY-CONNECTION is
   sorts Src, Dst
   op conn : Src, Dst -> Boolean
end-spec
```

`BINARY-CONNECTION` introduces the concept of a connection which has a source (`Src`) and a destination (`Dst`). Also introduced is the boolean operation `conn` which takes a source and destination as arguments and returns true if they are, in fact, connected together. Because `BINARY-CONNECTION` is intended to be used exclusively in other specifications, and not by itself, no axioms or operational definitions are included with it. Also, the sort `Boolean` is not explicitly defined, since it is built into Specware.

```
spec CONSTRAINED-BINARY-CONNECTIONS is
   import BINARY-CONNECTION
   op injective?:(Src, Dst -> Boolean) ->
                 Boolean
   op surject?  :(Src, Dst -> Boolean) ->
                 Boolean
   op funct?    :(Src, Dst -> Boolean) ->
                 Boolean
   op total?    :(Src, Dst -> Boolean) ->
                 Boolean
   definition of injective? Is
      axiom (iff (injective? conn)
        (fa (x:Src y:Src b:Dst)
            (implies(and (conn x b)
                         (conn y b))
                  (eq x y))))
   end-definition
   ...
end-spec
```

Next, a `CONSTRAINED-BINARY-CONNECTIONS` specification is created by importing the `BINARY-CONNECTION` specification. This type of inclusion is similar to the C `#include` preprocessor directive. `CONSTRAINED-BINARY-CONNECTIONS` contains the signatures and definitions for the four key multiplicity operations previously listed in Table 2. For brevity, only the injective? definition is shown. The original specification contains definitions for all four operations.

Using the operations contained in CONSTRAINED-BINARY-CONNECTIONS, a series of specifications can be created which define particular binary connections with multiplicity. For example, the ZO-to-ZM-CONN listed below defines a specification that allows zero-to-one source objects to be associated with zero-to-many destination objects. Such a connection was discussed in the previous section. ZO-to-ZM-CONN contains an op called zo-to-zm? which defines this relationship. As expected from the earlier discussion, this relationship is described mathematically as an injective relationship.

```
spec ZO-to-ZM-CONN is
    import CONSTRAINED-BINARY-CONNECTIONS
    op zo-to-zm? : (Src, Dst -> Boolean) ->
                      Boolean
    definition of zo-to-zm? is
        axiom (iff (zo-to-zm? c)
               (injective? c))
    end-definition
    axiom (zo-to-zm? conn)
end-spec
```

Although they specify relationships that are constrained with respect to multiplicity, specifications such as ZO-to-ZM-CONN are not useful in and of themselves. Instead, they are placed in a model composition constraints library, and are combined with other specifications to form a complete object interconnection specification.

Recall that in this example problem, the modeling paradigm requires connections involving AParts as sources and BParts as destinations to be of type zero-to-one to zero-to-many. This domain-specific requirement is described by the A-to-B-INTERCONNECTION specification listed below.

```
spec A-to-B-INTERCONNECTION is
   translate
      colimit of diagram
         nodes
           S:TRIV, D:TRIV,
           ZO-to-ZM-CONNECTION,
           APARTS, BPARTS
         arcs
           S -> APARTS : { E -> AParts },
           S -> ZO-to-ZM-CONN : { E -> Src },
           D -> BPARTS : { E -> BParts },
           D -> ZO-to-ZM-CONN : { E -> Dst }
      end-diagram
   by {conn -> ab-connection}
end-spec
```

Here, a colimit is formed between specifications that describe the endpoints of the interconnection (APARTS and BPARTS) and a specification that describes the constrained connection itself (ZO-to-ZM-CONNECTION). Two dummy specifications (S and D, both of type TRIV, which each contain a single sort, E) act as "glue points" during the colimit operation. The colimit creates a new specification from the union of specifications cited in the nodes section of the colimit. The arcs section of the colimit allows the specifier to indicate how the nodes are connected together, and which sorts from each node (i.e. specification) are associated with each other. Thus, the colimit can be seen as a union of specifications with selective sharing of sorts.

The first two lines in the arcs section state that the APARTS and ZO-to-ZM-CONN specifications are associated together, and that the sort AParts is associated with sort Source via the "glue" sort E. In other words, the source of this particular type of interconnection is an AParts type of object. Similarly, the last two lines in the arcs section identify the destination of the connection as a BPARTS object, by associating the sort BParts in the BPARTS specification with sort E of the D specification, which is also associated with sort Dst of the ZO-to-ZM-CONN specification. Although both the D and S specifications contain a sort named E, the E's are unique – sort E of specification D is distinct from sort E of specification S.

Finally, notice that the A-to-B-INTERCONNECTION refines the notion of a connection into the more specific "ab-connection." This is done by translating the conn operation, which originated in the BINARY-CONNECTION specification, into an operation called ab-connection.

This example has shown that it is possible to use a formal specification language to create general model composition constraint specifications and refine and compose them into a domain-specific specification. Such an exercise is useful in its own right, to establish the credibility of a modeling paradigm, to allow formal reasoning of a specification, and to document the system's design in a formal way. However, this is not enough. The MIC environment designer must be able to take the formal specification, i.e. the meta model, and translate it for use in synthesizing the MIC environment. This final phase of the work is discussed below.

Mediator and MDF generation

Only by generating an MDF can a modeling paradigm's formal specification be fully utilized. Generating an MDF from a formal specification requires detailed knowledge about the application programming interface (API) used by the specification language, so that key information may be extracted from the final meta model specification and used to generate the MDF.

Specware was written using Refine [8]. (Refine is a programming environment for language design which uses BNF-like notational descriptions of the language's grammar. Refine is a product of Reasoning Systems of Palo Alto, CA, USA.) Because of this, a Refine-based *mediator* had to be written to examine the resulting Specware data structures, extract certain information necessary to generate an MDF, and to finally create the MDF. Because Specware is an emerging technology, little formal documentation was available on the API used to access the underlying data structures. However, with the help of Specware design and maintenance personnel at the Kestrel Institute, a mediator was developed to generate MDF files from simple specifications – simpler specifications than the example just presented. (The modeling paradigm was the same, but the source specifications contained only carefully written axioms from which the MDF information could be easily extracted.) The generated MDF was then used with the MGA to synthesize a modeling environment which conformed to the specified modeling paradigm. A more detailed mediator, capable of creating an MDF from the specifications presented in the proof-of-concept example contained in this paper, was planned, but not completed. Nonetheless, the concept of an automatically generated MDF was demonstrated, albeit on a small scale.

Conclusions and future work

Clearly the need exists to apply meta modeling techniques to the Multigraph MIPS environment, and the approach discussed in this paper appears viable. The key to meta modeling in this application is the use of a formal specification language. Formal specification languages enable system specifications to be standardized, allow reuse of general modeling concept specifications, and enable the MIPS environment creator to reason about the modeling paradigm, validating the meta model's consistency before creating an actual MIPS environment.

Building on the work of Bourdeau and Cheng, this paper develops Specware specifications for the model composition constraints of binary object association and connection multiplicity. These formal specifications were then used in a proof-of-concept example to demonstrate refinement of general model composition constraint specifications into a domain-specific meta model. A Specware mediator was created which generated an MDF automatically from a simple set of specifications.

The remaining model composition constraints (hierarchy, aspects, and specialization) must now be formalized. However, new formal specification languages should be investigated, such as PVS or Larch, as an alternative to Specware. Larch appears particularly attractive due to its open nature and the ability to create domain-specific interface language specifications. Regardless of the language chosen, it must allow easy access to key specification information in order to allow an MDF (or an MDF-like file) to be automatically generated for use by the MGA to synthesize domain-specific MIPS environments. Because the current MDF language is known to be inadequate for specifying the broad range of MIPS environments required, the development of meta level extensions to the MGA must be accompanied by, and must be done in conjunction with, an effort to upgrade and expand the current model description language used in the MGA.

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. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Sorenson, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[2] *UML Summary*, Rational Software Corporation, 1997

[3] Y.V. Srinivas and R. Jüllig, *About Specware*, Suresoft, Inc., 1996

[4] R. Waldinger, Y.V. Srinivas, A. Golberg, R. Jüllig, *Specware Language Manual*, KDC and Suresoft, Inc., 1996

[5] J.V. Guttag and J.J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[6] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. *PVS: Combining specification, proof checking, and model checking*. In Rajeev Alur and Thomas A. Henzinger, editors, Computer-Aided Verification, CAV '96, volume 1102 of Lecture Notes in Computer Science, pages 411-414, New Brunswick, NJ, Springer-Verlag, July/August, 1996

[7] R.H. Bourdeau and B.H. Cheng, *A Formal Semantics for Object Model Diagrams*, IEEE Transactions on Software Engineering, Vol. 21, No. 10, October, 1995

[8] *Refine User's Guide*, Reasoning Systems Inc., 1990