Metaprogrammable Toolkit for Model-Integrated Computing

Akos Ledeczi, Miklos Maroti, Gabor Karsai and Greg Nordstrom
Institute for Software Integrated Systems
Vanderbilt University

Abstract

Model-Integrated Computing, specifically Model-Integrated Program Synthesis (MIPS) environments that include visual model building, constraint management, and automatic program synthesis components, are well suited for the design and implementation of complex computer-based systems. However, building such an environment from scratch for each new domain can be cost-prohibitive. This paper presents a toolkit that makes the rapid creation of MIPS environments possible through metaprogramming.

Introduction

Complex computer-based systems are characterized by the tight integration of information processing and the physical environment of the systems. Model-Integrated Computing (MIC) is well suited for the rapid design and implementation of such systems. MIC employs domain-specific models to represent the software, its environment, *and* their relationship. With Model-Integrated Program Synthesis (MIPS), these models are then used to automatically synthesize the embedded applications and generate inputs to COTS analysis tools. This approach speeds up the design cycle, facilitates the evolution of the application and helps system maintenance, dramatically reducing costs during the entire lifecycle of the system.

Creating domain-specific visual model building, constraint management, and automatic program synthesis components for a MIPS environment for each new domain would be cost-prohibitive for most domains. Applying a generic environment with generic modeling concepts and components would eliminate one of the biggest advantages of MIC – the dedicated support for widely different application domains. An alternative solution is to use a configurable environment that makes it possible to customize the MIPS components for a given domain.

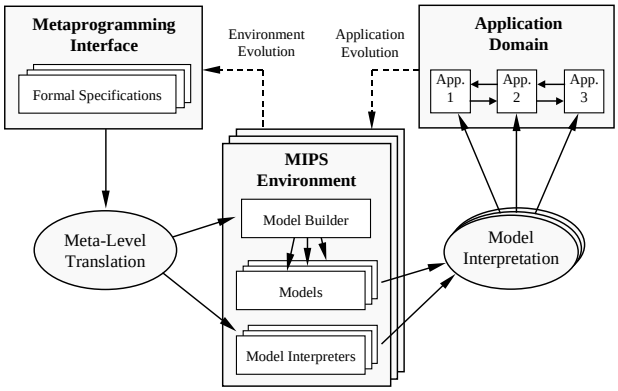

Figure 1: The Multigraph Architecture

The Multigraph Architecture (MGA), being developed at the Institute for Software Integrated Systems at Vanderbilt University, is a toolkit for creating domain-specific MIPS environments. The MGA is illustrated in Figure 1. The metaprogramming interface is used to specify the modeling paradigm of the application domain. The modeling paradigm is the modeling language of the domain specifying the modeling objects and their relationships. In addition to syntactic rules, semantic information can also be described as a set of constraints. The Unified Modeling Language (UML) and the Object Constraint Language (OCL), respectively, are used for these purposes in the MGA. These specifications, called metamodels, are used to automatically generate the MIPS environment for the domain. An interesting aspect of this approach is that a MIPS environment itself is used to build the metamodels. Furthermore, metamodels are created to describe this MIPS environment. These are called the meta-metamodels. A separate paper [2] describes the metaprogramming aspects of the MGA in more detail.

The generated domain-specific MIPS environment is used to build domain models that are stored in a model database. These models are used to automatically generate the applications or to synthesize input to different COTS analysis tools. This process is called model interpretation.

Model Builder Architecture

Model building is the creation of representations that (1) describe an artifact using a certain formalism and (2) satisfy constraints that capture the semantics of the domain. The MGA supports a primarily graphical formalism for modeling. It provides a rich set of graphical idioms for the metamodeler to choose from to implement the entities and relationships of the application domain. (Textual representations are also supported when necessary.) An MGA modeling paradigm always contains a set of explicit constraints expressed in the Object Constraint Language (OCL). These define the static semantics of the domain. Note, however, that selecting a given graphical idiom introduces additional (i.e. implicit) constraints into the modeling paradigm. For example, a simple connection can only express an association between two objects.

The structure of a model builder is determined by the services it needs to provide:

- an interface to the human modeler using a well-defined syntax (graphical, tabular, textual),
- operations that manipulate the models,
- a constraint manager that checks and/or enforces the semantics of the domain,
- a model database with access mechanisms, and
- an interface for model interpreters to access and/or manipulate the models.

Metaprogramming configures (1) the metaprogrammable model builder by assigning semantics to the graphical idioms it provides and (2) the constraint manager by assigning constraints (semantics) to the operations. The overall architecture of an MGA MIPS environment is depicted in Figure 2.

At the core of the MGA MIPS environment is the Graphical Model Editor (GME). This component is responsible for maintaining the model structures and providing the operations to manipulate them. The GME coordinates with the other components as well. Component integration is done using the Component Object Model (COM).

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

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

The Constraint Manager has access to the set of domain-specific constraints provided by the Paradigm Definition module and the models maintained by the GME. Checking of selected constraints can be triggered by certain requested operations, such as *connec*t, *close*, or *modify Attribute*. All constraints can be checked at once upon explicit request. In addition to the triggering condition, a constraint also has a context, a priority, and an OCL expression associated with it.

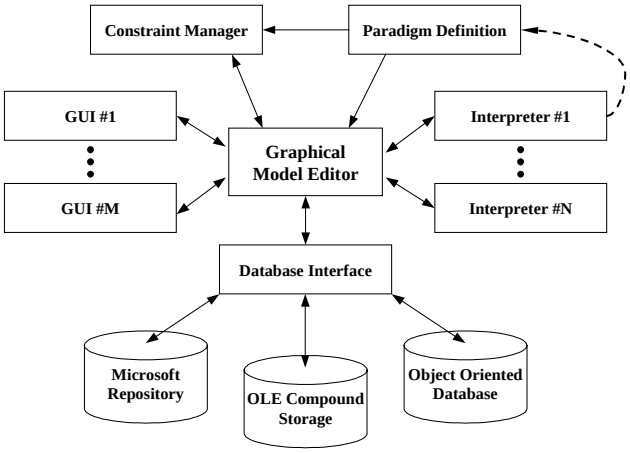

Figure 2: Model Builder Architecture

The modeling paradigm definition module contains the configuration of the graphical idioms and operations that constitute the given paradigm. It is directly generated by the metaprogramming layer of the MGA. Since that layer is implemented by the same MIPS infrastructure, Figure 2 can be interpreted as the metaprogramming environment for generating domain-specific MIPS environments. If the models describe the metaprogramming environment (i.e. in case of the meta-metamodels) then the toolkit configures itself (i.e. is self-booting). The dashed line from one of the model interpreters to the paradigm definition module illustrates this concept.

Model interpreters translate the system models into executable applications or input to COTS analysis tools. Model interpreters are domain-specific. Currently they are not configured automatically from the paradigm description. However, formally specifying the mapping between the model objects and the runtime objects and automatically generating the interpreters is an active research area at the ISIS [6].

The GME provides a COM interpreter interface that supports the access and modification of the models. It also provides visualization hooks, so interpreters are able to provide feedback to the user through messages displayed in the correct context. An extension to this interface that will support animation is planned. Implemented on top of the COM interface, there is a high-level, extensible, C++ interpreter interface. This interface provides all the common tasks required for model interpreters, significantly reducing the time and effort required when writing model interpreters.

Modeling Concepts

The metaprogrammable model builder works with the following concepts:

- Paradigms,
- Categories,
- Atoms,
- Models,
- Ports,
- Aspects,
- Attributes,
- Hierarchical containment,
- Connections,
- References, and
- Conditionals.

Figure 3 illustrates the complex relationships among these constructs. The *Paradigm* defines the entities and relationships allowed in the given domain. Related models are grouped into *Categories*. Each Category has its own model hierarchy. Each Paradigm has a fixed set of Categories. For example, in the parallel instrumentation domain we could have a signal flow Category containing the hierarchical signal flow of the application, and a hardware Category describing the topology of the parallel DSP network [4].

The basic modeling objects are *Atoms* and *Models*. Atoms are the elementary objects – they cannot contain parts. Each kind of Atom is associated with an icon and can have a predefined set of attributes. The attribute values are user changeable. A good example for an Atom is an AND or XOR gate in a gate level digital circuit model.

Models are the compound objects in our framework. They can have parts and inner structure. The modeling paradigm determines what *kind* of parts are allowed in Models, but the modeler determines the specific type and number of parts a given model contains (of course, constraints can always restrict the design space). For example, if we want to model digital circuits below the gate level, then we would have to use Models for gates that would contain transistor Atoms.

This containment relationship creates the hierarchical decomposition of the Models. If a Model can have the same kind of Model as a contained part, then the depth of the hierarchy can be (theoretically) unlimited. Any object must have at most one parent, and that parent must be a Model. At least one Model does not have a parent, it is called a *root Model*. A good example for this containment hierarchy is a modeling paradigm for the production flow of discrete manufacturing, such as a car assembly plant. The top-level (i.e. root) process Model corresponds to the whole plant. This Model contains Models corresponding to different parts of the plant, such as powertrain and body systems. These in turn contain sub-process Models and so on, all the way down to the machine level [5]. Hierarchy is an effective method for controlling the complexity of the models themselves.

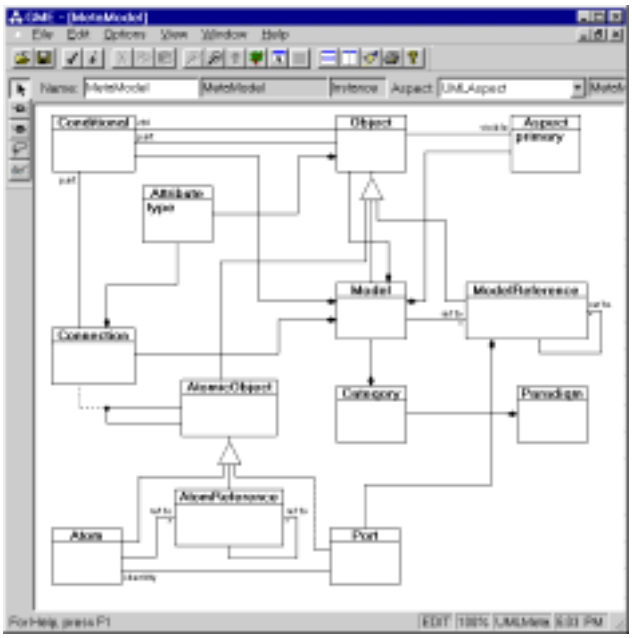

Figure 3: Modeling Concepts

The Paradigm can specify that instances of certain kinds of Atoms appear on the outside interface of the container model as Ports. The primary purpose of Ports is to enable making Connections to Models.

Aspects provide primarily visibility control. Every Model has a predefined set of Aspects. Each part (and Connection) can be visible or hidden in an Aspect. Every part (and Connection) has a primary aspect where it can be created or deleted. The set of Aspects of a Model must be a subset of the set of Aspects of any one of its parts. In other words, a part must have the same Aspects as its parent but it can have extra aspects as well.

The simplest way to express a relationship between two objects in the MGA modeling environment is with a Connection. Connections can be made between Atoms, Atom References (explained later), and Ports. A Model cannot be connected directly, only through one of its Ports. Connections can be directed or undirected. Connections can have Attributes themselves. In order to make a Connection between two objects they must have the same parent in the containment hierarchy (and they also must be visible in the same Aspect, i.e. the primary Aspect of the Connection). The paradigm specifications can define several different kinds of Connections. It is also specified what kind of object can participate in a given kind of Connection. The signal flow paradigm provides a good example. Signal flow Models contain input- and output signal Atoms and they appear as input- and output Ports on their outside interface. Signal flow Connections can only be created between input- and output signals and input- and output ports. Connections can further be restricted by explicit Constraints specifying their multiplicity, for instance.

A Connection can only express a relationship between objects contained by the same Model. Note that a Root Model, for example, cannot participate in a Connection at all. In our experience, it is often necessary to associate different kinds of model objects in different parts of the model hierarchy or even in different model hierarchies (Categories) altogether. *References* support these kind of relationships well.

References are similar to pointers in object oriented programming languages. A *reference* is not a "real" object, it just refers to (points to) one. In GME, a reference must appear as a part in a Model. This establishes a relationship between the Model that contains the reference and the referred object. Atoms, Models and references themselves can be referred to. References can be connected just like regular model objects. Atom (and Atom reference) references can be connected directly. Model (and Model reference) references get copies of the Ports of the referred Model. These Ports can then participate in Connections. A reference always refers to exactly one object, while a single object can be referred to by multiple references.

Connections and references model relationships between at most two objects. *Conditional*s can be used to express association among two sets of objects. The first set is the so-called Controller. This set must consist of the same kind of model objects (Atoms, Models or references). Usually the Controller set has a single element. The items in the second set are called the parts of the Conditional. These can be of several different kinds of objects and Connections (defined in the paradigm specifications). Both of the sets must have at least one element.

The name Conditional comes from the most typical use of this modeling construct. We can describe a dynamic system by modeling it with a state machine and associate the states with parts of the system models. In this case, the states are the Controller objects of the Conditionals and the parts are the model objects and connections that are present in the given state.

Being a complex modeling construct, the visualization and creation of Conditionals are somewhat complicated. In the Conditional Mode of the GME, the user must select the Controller object(s) first. All the model objects and connections that are not part of the Conditional with the selected set of controls are grayed out at this point. The user can then add or remove parts by simply clicking on them. This means that for one kind of Conditional there can be at most one instance with the same set of Controllers. Another restriction is that all the Controllers and parts of a Conditional must have the same parent and must be visible in the same Aspect.

Some kinds of information do not lend themselves well to graphical representation. The GME provides the facility to augment the graphical objects with textual attributes. Most modeling objects (Atoms, Models, References, and Connections) can have a set of Attributes. The kinds of Attributes available are text fields (string type), multi-line text areas (string type), toggle switches (boolean type), and menus (integer type).

Graphical Model Editor

There are two ways to achieve metaprogrammability in the context of a MIPS environment. The first approach is to automatically generate the code of the different modules of the environment from the metamodels. The other is to have generic, i.e. paradigm-independent, modules that are able to configure themselves from the metamodels (or, more precisely, from the paradigm definition, which is an intermediate representation generated from the metamodels). MGA uses the latter approach, sacrificing efficiency to a small extent for added flexibility.

The GME, its native graphical user interface, the Constraint Manager, the database interface, and the COM and high-level interpreter interfaces are all paradigm-independent modules that configure themselves on-the-fly at runtime. This approach speeds up the design cycle of domain-specific MIPS environments. The metamodeler can edit the metamodels, generate the paradigm specification with the appropriate interpreter, and without exiting the environment load the generated paradigm and begin building models. Designing a modeling paradigm is an inherently iterative process, so speeding up this cycle can result in large productivity increase. The price to pay for this flexibility is a slight loss of efficiency.

To illustrate this point, consider a model with an integer attribute for priority. A generated model builder would have a data structure for this model with an integer field for the priority. A paradigm independent tool, on the other hand, would have a list of generic attributes, each with a description of its type and value. Priority would be one element in this list. This requires more storage and slower access. However, a careful implementation can minimize these effects.

Graphical User Interface

The native graphical user interface of GME is shown in Figure 4. The picture shows the GME with a signal flow paradigm and a simple mode loaded. It shows Atoms, Models, Ports and Connections in the main window, the hierarchical decomposition of the Models in the Model Browser window and textual attributes in the Attributes windows. Currently the native user interface is integrated into the GME. Additional user interfaces are supported through the interpreter interface.

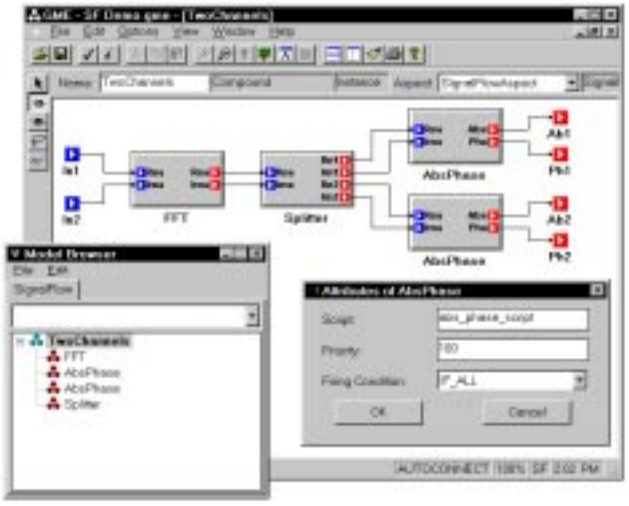

Figure 4: The Graphical Model Editor

Interpreter Interface

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

An interpreter can be implemented as either a dynamic link library (DLL) or executable module. Using DLLs will result in in-process activation, and consequently, negligible overhead. Executable interpreters run considerably slower but have the added flexibility of being able to initiate interpretation independently from the GME. Interpreter DLLs are registered in the Windows registry for the different paradigms. The actual DLL is not loaded until the user requests interpretation from the GME and selects the desired interpreter (in case there are multiple interpreters registered for the current paradigm).

Despite its flexibility, the COM interface is still quite low-level. For anything but very simple interpreters, the interpreter writer has to create complex data structures and traverse the model hierarchy several times to build them up. Our experience shows that there is a large set of common steps among different interpreters in significantly different paradigms. The high-level, C++ interpreter interface implements these phases of the interpretation creating a layer above the COM interface. Figure 5 illustrates the interpreter interface architecture.

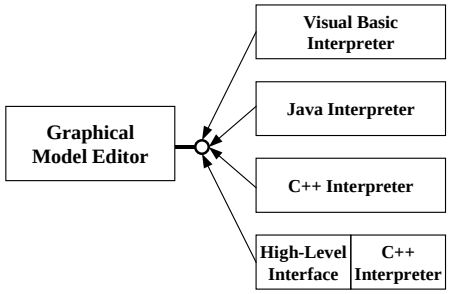

Figure 5: The Interpreter Interface

The high-level interpreter interface defines its own generic classes and builds up a so-called builder object network whose structure closely resembles that of the models. However, the builder objects facilitate traversing the models along the hierarchy, connections, or references much better. For example, it allows following a connection from leaf to leaf bypassing the hierarchy with a single function call.

The builder object network is already built by the time the interpreter receives control. The interface is easily extensible: the built-in builder classes can be extended with inheritance. The interpreter interface will automatically instantiate the user-defined classes.

Constraint Manager

The Constraint Manager is presented with a domain-specific constraint set. A constraint consists of a model context definition, a triggering event, a name, an optional argument list, a priority, and an OCL expression. In the following example, the constraint states that all the parts of every model in the SignalFlow paradigm must have a unique name:

```
in SignalFlow.Paradigm on demand_event
   constraint UniqueNames () priority = 3
   "Models must have unique names"
{
   parts()->forAll(p1, p2 |
      p1 <> p2 implies
                  p1.name() <> p2.name())
}
```

The context specifies which Category, Model, and Aspect must satisfy the given constraint. The triggering event specifies when a given constraint needs to be checked. For example, the existence of a given part would likely be checked at the time the container Model is being closed. A constraint on a Connection should be checked when that type of Connection is created. Constraints need not be tied to any given event. All the constraints can be evaluated upon demand by the user, as indicated by the "demand_event" specification in the above example.

The priority is used to control the order in which constraints are checked. The constraint manager starts by evaluating the higher priority constraints, enabling the user to use an iterative approach to fixing possible violations (notifications arrive from the constraint manager as soon as a violation is found). This avoids generating a long list of constraint violation messages such as some compilers do when in fact a single syntax error exists.

The description field is used to provide feedback to the user after a constraint violation. The OCL expression specifies the actual constraint. A predefined set of functions allows access to parts, attributes, etc., of the models.

Conclusions

The ability to rapidly create domain-specific Model-Integrated Program Synthesis environments makes Model-Integrated Computing a cost effective approach to a wide range of applications. The key characteristics of a metaprogrammable MIPS environment are a rich set of graphical formalisms, a powerful constraint management component, and an extensible, modular architecture. Generic components that are able to configure themselves according to the given modeling paradigm provide enormous flexibility. The MGA is gaining widespread acceptance in the engineering community. MGA applications are being actively used in different domains such as Saturn, Boeing, NASA, USAF, and Sandia.

Acknowledgements

This work was sponsored in part 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., et al.: "MULTIGRAPH: An Architecture for Model-Integrated Computing," *Proceedings of the IEEE ICECCS'95*, pp. 361-368, Nov. 1995.

[2] Nordstrom, G., et al.: "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments," *Proceedings of the IEEE ECBS'99 (in review)*, March 1999.

[3] Karsai, G., et al.: “A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming,” Computer, March 1995.
[4] Ledeczi, A.: “Model-Integrated Parallel Application Synthesis,” *Proceedings of the IEEE ECBS'97*, March 1997
[5] Misra, A., et al.: “A Model-Integrated Information System for Increasing Throughput in Discrete Manufacturing,” *Proceedings of the IEEE ECBS'97*, March 1997
[6] Karsai, G., et al.: “Automatic Model-Interpreter Generation,” *Proceedings of the IEEE ECBS’99 (in review)*, March 1999.