In the Proceedings of the 13th International Conference on Systems Engineering Las Vegas NV, August 1999

MODEL CONSTRUCTION FOR MODEL-INTERGRATED COMPUTING

Akos Ledeczi
Vanderbilt University
Nashville, TN 37235
akos@isis.vanderbilt.edu

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 input to system 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 graphical 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.

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.

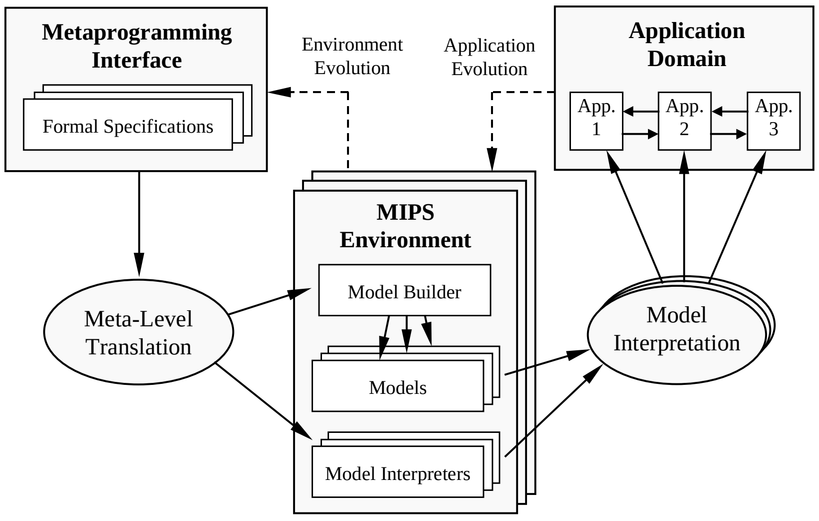


Figure 1: The Multigraph Architecture

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.

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.

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 at diverse places. The Saturn Site Production Flow system monitors the manufacturing process and provides key production measures to managers in real-time [1]. The system models describe the manufacturing processes down to the machine level, the buffers between the processes, the instrumentation (i.e. PLCs), and how the information is to be presented to the user. The model interpreters generate different configuration files and SQL database schema to configure the SSPF client-server application. The program gathers the production information, stores it in a real-time database and makes it available to any user in the plant. Within months of the start of production use of the system, the Spring Hill site of Saturn Corporation reported a 10% productivity increase. A second installation in Delaware took a couple man-weeks of modeling time and two hours of installation. There was no need for any code modification illustrating the power of MIC.

Other MIC applications include a system in production use by Boeing and NASA for Fault Detection, Isolation and Recovery for the International Space Station [3], a tool used by Sandia National Labs for system safety and surety [2], an integrated test information system at USAF Arnold Engineering Development Center and a real-time vibration analysis system being used by NASA for Space Shuttle Main Engine testing and by the USAF for jet engine testing [4]. Several ongoing DARPA funded research projects are under way to extend the capabilities of the Multigraph Architecture [5-6].

Model Construction

Model construction is one of the key steps when creating an application using MIC. 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 graphical idioms introduce additional (i.e. implicit) constraints into the modeling paradigm.

The metaprogrammable model builder works with the following concepts: Paradigms, Categories, Atoms, Models, Ports, Aspects, Attributes, Hierarchical Containment, Connections, References, and Conditionals. Figure 2 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.

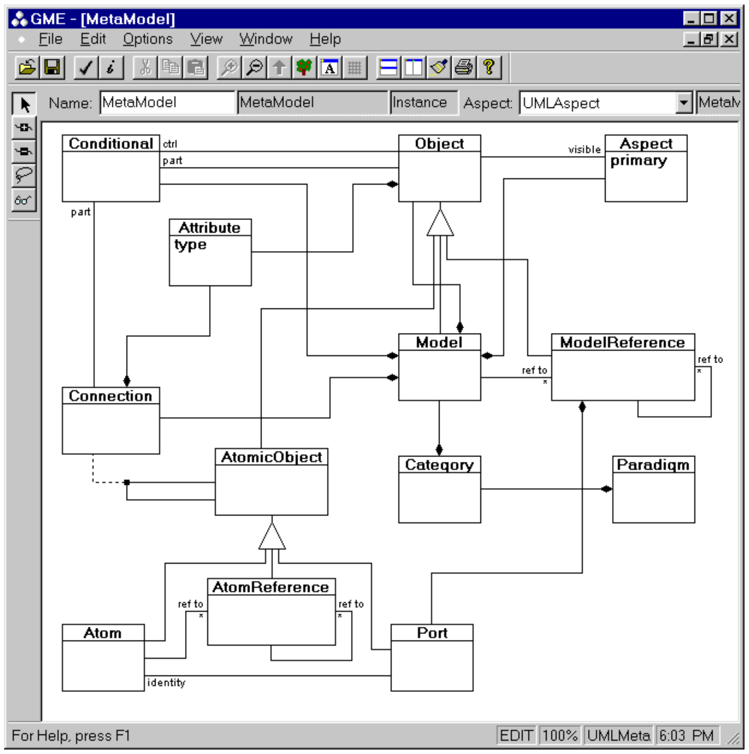


Figure 2: Modeling Concepts

The basic modeling objects are *Atoms* and *Models*. Atoms are the elementary objects – they do not 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 NAND or OR 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 [1]. Hierarchy is an effective method for controlling the complexity of the models themselves.

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 this 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 MGA, a reference must appear as a part in a Model. This establishes a relationship between the Model that contains the reference and the referred to 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.

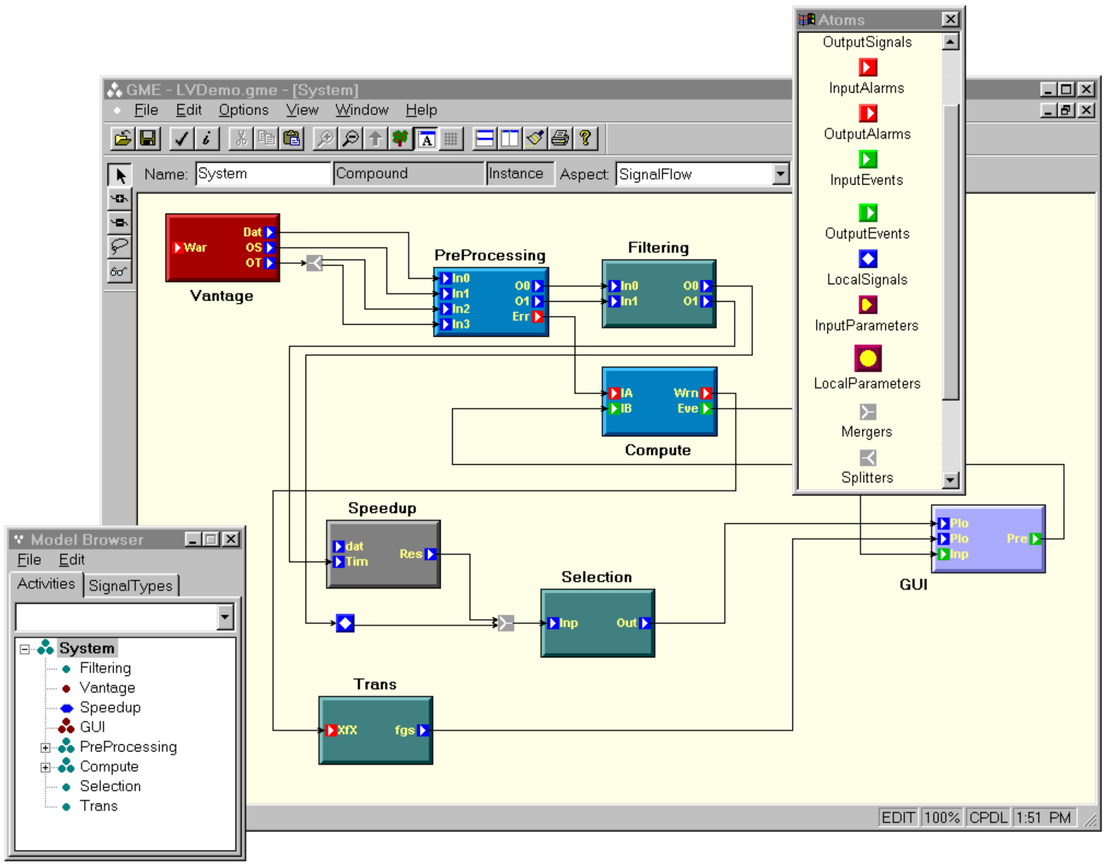
Connections and references model relationships between at most two objects. *Conditional*s can be used to express association among two sets of objects. The first set is the so-called Controller. This set must consist of the same kind of model objects (Atoms, Models or references). Usually the Controller 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. A restriction similar to that of Connections 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 MGA 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, multi-line text areas, toggle switches, and menus.

Example Application Domain

Figure 3 displays an example model using the modeling paradigm developed for the Activity Modeling Tool (AMT), a software environment for process monitoring in chemical plants currently being developed for the DuPont DMT Plant in Old Hickory, Tennessee. The AMT makes it possible to rapidly create custom monitoring and simulation applications. The user models what data points she needs from the Vantage real-time database, what kind of processing needs to be done, what input and output Speedup, a COTS external simulation package, needs, and how the user interface should look like. The custom application is then automatically synthesized from these models and different libraries including the Vantage interface-, the Speedup interface-, the configurable GUI-, a dataflow kernel-, and others libraries that may contain any additional application-specific code. The output of this translation process are different configuration files and C++ code which is automatically compiled and linked together with the above libraries.


GME - LVDemo.gme - [System]

Name: System Compound Instance Aspect: SignalFlow

Figure 3: Modeling Example

Some of the modeling concepts are illustrated by the above figure. The AMT paradigm has two categories of models: Activities and Signal Types. The models browser in the lower left hand corner displays the categories with separate tabs. Each category has its own hierarchy of models. The window in the upper right hand corner displays the set of atoms that are available in the current aspect of the current model, i.e. the active model in the main window. The Signal Flow aspect of the Compound model called System is shown in the large window in center of Figure 3. In this model, you can see examples for atoms, models, ports and connections.

Conclusions

Model-Integrated Computing (MIC) is well suited for the rapid design and implementation of complex computer-based systems characterized by the tight integration of information processing and the physical environment. Using a configurable MIC environment, that makes it possible to inexpensively customize the components for a given domain, provides a cost-effective solution. In the Multigraph Architecture (MGA), the same exact modeling environment is used in widely different domains, including even the metamodeling domain [7], as illustrated by Figures 2 and 3. The rich set of general graphical modeling formalisms and the flexible constraint management make the MGA fit a wide range of application domains.

References

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

2. Davis J., Scott J., et al.: "Integrated Analysis Environment for High Impact Systems," *Proceedings of the Engineering of Computer Based Systems Conference*, Jerusalem, Israel, April, 1998

3. Carnes J. R., Misra A.: "Model-Integrated Toolset for Fault Detection, Isolation and Recovery (FDIR)", *Proceedings of the International Conference and Workshop on Engineering of Computer Based Systems*, Friedrichshafen, Germany, March, 1996.

4. Abbott B., Bapty T., et al.: "Model-Based Approach for Software Synthesis", *IEEE Software*, May, 1993

5. Sztipanovits J., Karsai G., Bapty T.: “Self-Adaptive Software for Signal Processing,” *Communications of the ACM,* Vol. 41, No. 5, pp. 66-73, May, 1998.

6. Bapty T., Scott J. et al.: "Uniform Execution Environment for Dynamic Reconfiguration," *Proceedings of the Engineering of Computer Based Systems Conference*, Nashville, TN, March, 1999

7. Nordstrom G. et al: "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments," *Proceedings of the Engineering of Computer Based Systems Conference*, Nashville, TN, March, 1999