Model Integrated Computing: A Framework for Creating Domain Specific Design Environments

James R. DAVIS
Vanderbilt University, Institute for Software Integrated Systems
Nashville, TN 37203, USA

ABSTRACT

Model Integrated Computing (MIC) is a technology developed to aid in the rapid design and implementation of complex computer based systems. These systems typically are characterized by the integration of their information processing systems and the physical environment of the actual system. MIC employs multiple aspect, domain-specific modeling technology to represent the system software, the system hardware, its environment, *and* their relationships. Model interpreters are used to transform the information captured in the models into the artifacts required by the chosen analysis tools or run time system. One of the largest advantages to using MIC is the ability to reason and design a complex system at a higher level of abstraction. This paper will describe one framework for applying MIC to system tool design. A selected project where the framework is being applied will be introduced. The advantages to using MIC for this project will be discussed.

Keywords: modeling, model translation, modeling language specification, simulation integration

1. Introduction

Complex computer-based systems are often characterized by the tight integration of information processing and the physical environment of the actual system. In addition, these systems are often mission critical systems; their failure is unacceptable. Model-Integrated Computing (MIC) is a technology that is well suited for the rapid design, implementation, and evolution of such systems [1]. MIC employs domain-specific models to represent system software, its environment, *and* their relationships. With Model-Integrated Program Synthesis (MIPS), these models are then used to automatically synthesize the embedded software and hardware applications and generate inputs to COTS analysis tools. The MIPS technique is possible only due to the capturing of the relationships between the software and the system's environment. 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.

2. Model Integrated Computing

The Multigraph Architecture (MGA), developed at the Institute for Software Integrated Systems at Vanderbilt University, is a toolkit for creating multiple aspect, domain-specific MIPS environments. The MGA is shown in Figure 1. The metaprogramming interface is used to formally specify the application domain's modeling paradigm. The modeling paradigm captures all the syntactic, semantic, and presentation 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 modeling environment. All modeling paradigms additionally adhere to a set of specifications regarding the presentation features allowed by the MGA configurable model editor.

With MIC, modeling paradigms are represented by metamodels. The metamodels are used to automatically configure the MIPS modeling environment for the domain. This MIPS environment consists of a *domain specific* model editor, a customized model database, and a set of model translators or interpreters. An interesting aspect of this approach is that a MIPS environment itself is used to build the metamodels [2].

The generated domain-specific MIPS environment is then used by the system user 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 interpreters are those entities that automatically translate the models into other useful artifacts while ensuring the semantics between the modeling domain and external tools are kept consistent.

The Generic Modeling Environment

The Generic Modeling Environment (GME 2000), is a Windows-based, domain-specific, model-integrated program synthesis tool for creating and evolving domain-

specific, multi-aspect models of computer based engineering systems. The GME 2000 is part of the Multigraph Architecture (MGA) tool suite. In particular, GME 2000 provides the domain specific model editor that is used in the MGA systems [3].

The GME 2000 is configurable, or meta-programmable, which means it can be "programmed" to work with vastly different domains. Another important feature is that GME 2000 is configured from formal modeling environment specifications or meta-models. This ensures that it can be quickly and safely evolved as modeling requirements change [4]. GME 2000 is used primarily for model-building. The models take the form of graphical, multi-aspect, attributed entity-relationship diagrams. The static semantics of a model are specified by explicit constraints that are enforced by a built-in constraint manager. The dynamic semantics is not the concern of GME 2000 – that is determined later during the model interpretation process.

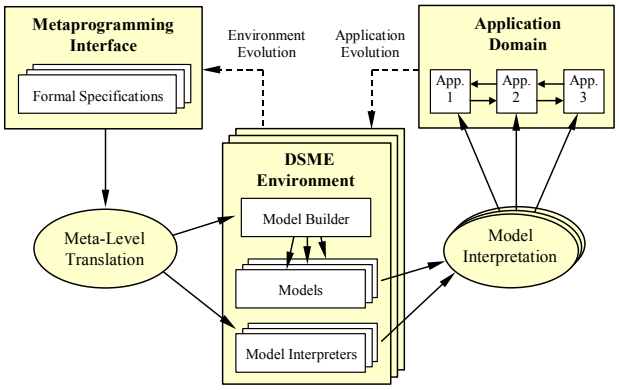

Figure 1 : The Multigraph Architecture

Modeling Concepts: The GME 2000 supports various techniques for building and managing the complexity of large-scale, complex models. The techniques include: hierarchy, multiple aspects, sets, references, and explicit constraints. The GME 2000 users manual [4] details the different relationships between the major modeling components. A brief overview of the general concepts will be given here.

Models are the centerpieces of a MIC environment. They are compound objects that can have parts and inner structure. Models can contain other models, atoms (parts that cannot be further decomposed), sets, references, and connections. Notice that since models can contain other models, hierarchical systems can be constructed. Textual attributes can be attached to most GME objects. This allows for capturing information that cannot be efficiently modeled graphically.

Associations between objects are captured using Connections, References, and Sets. Connections and References model relationships between at most two objects. References are used to associate objects in another part of the model hierarchy. Sets can be used to specify a relationship among a group of objects. The only restriction is that all the members of a set must have the same parent and be visible in the same Aspect.

Another key feature of GME 2000 is the ability to partition the models visually using Aspects. Using multiple aspects grants the ability to hide part of the modeled information from certain classes of users. Every Model has a predefined set of Aspects. Each component can be visible or hidden in an Aspect. Every component has a set of primary aspects where it can be created or deleted. There are no restrictions on the set of Aspects a Model, and it's parts, can have; a mapping can be defined to specify what Aspects of a part is show in what Aspect of the parent Model. A specific class of user may only want to see objects in the model that pertain to hardware. By carefully crafting the modeling language, the tool designer can allow this behavior.

When a particular type of model is created in a GME 2000 domain, it becomes a type (class). It can be sub typed and instantiated as many times as the user wishes. Please see [4] for more information about sub-typing with GME 2000. One, often confusing, issue is that the concept of the Model is one level higher in the meta hierarchy than that of the class in an OO language. A particular kind of Model in a modeling paradigm is equivalent to the concept of the class. In the resulting environment, the end user of the MIC system can create specific instances of the Model, which is similar to instantiating a class in an OO language.

It is important to note that when using GME 2000, the user deals with components in their domain. They do not need to understand models, atoms, references, etc. Instead, they need to understand how to use the features of their paradigm to construct models for their domain. In one of our projects [5], the users constructed models of a discrete manufacturing plant as a process model. The users dealt with processes, buffers, and conveyers; they did not deal with abstract models and atoms. A large part of the power of using MIC comes from the customization of the tools to a particular problem domain.

Interfacing to GME 2000

GME 2000 has a modular, Microsoft COM-based architecture depicted in Figure 2. Details of the different components are outside the scope of this paper. Two important components that will be discussed here are the *Add-On* and *Interpreter*.

The MGA and Meta components expose a set of COM interfaces that can be used to write model interpreters and add-ons. The GME 2000 user interface has its own COM interface that supports program-driven visualization of models. Notice that all GME 2000 components interface through the use of the MGA and Meta component COM interfaces. Through these interfaces, the user can write interpreters and add-ons that access the model information and provide some type of translation.

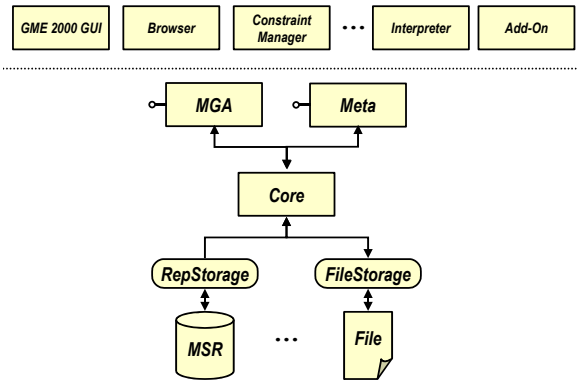

Figure 2: GME 2000 Architecture

In addition to these COM interfaces, GME 2000 provides an interface for non-COM programmers. A high-level component interface sits on the top of the MGA and Meta COM interfaces and provides a C++ API for interpreter writing. It implements a set of C++ classes that are instantiated immediately upon interpreter invocation. A network of objects (called the Builder Object Network) is built that mirrors the structure of the whole project before the interpreter gets control. It is important to note that the whole project is mirrored – for potentially very large projects, the native COM interpreter interface is preferred. The high level interface unburdens the user from making relatively low-level COM calls. The user can use these services through the public interfaces of the C++ objects.

This interface can be extended using C++ inheritance. The user can derive from the built-in classes and the interface will automatically instantiate the user-defined classes instead of the built-in ones using the object factory design pattern. In a graduate-level course on MIC, the extension of the BON is stressed as almost essential for complex projects.

Interpreters are the model translators discussed earlier in this paper. They are executed on demand, take the models as input, and deliver some type of output based on the models. One can think of the model interpreters as *applying* the semantics to the models.

Add-ons can be considered event-driven model interpreters. A set of events, such as "Object Deleted", "Set Member Added", and "Attribute Changed" are exposed by lower level GME components. External components can register for a set of these events. They are automatically invoked by the GME 2000 components whenever the events occur. Add-ons are generally used for extending the capabilities of the GME User Interface. When a particular domain calls for some special operations, they can be supported without requiring the modification of GME 2000. This architecture is very flexible and supports extensibility of the entire environment. The GME 2000 Users Manual provides detailed documentation on the high-level component interface [4].

3. Domain Specific Language Specification

Defining a domain specific modeling paradigm is itself a problem domain. Metamodeling is a term used to describe the process of modeling the domain specific modeling language. Semantics, syntax, and presentation are all captured in the metamodel. It is quite natural that GME 2000 is used to construct these modeling language models, or metamodels.

There is a metamodeling paradigm defined that configures GME 2000 for creating metamodels. These models are then automatically translated into GME 2000 configuration information through the model interpretation process. Originally, the metamodeling paradigm was handcrafted. Once the metamodeling interpreter was operational, meta-metamodels were created and the metamodeling paradigm was generated automatically. This is similar to writing a C compiler in C. Note that meta-metamodels is the point where the meta hierarchy ends. Since we use the metamodeling environment itself to create the meta-metamodels, there is no need for an additional level; there are no meta-meta-metamodels [2, 6].

The metamodeling paradigm is an extension of the Unified Modeling Language (UML). In fact, the syntactic definitions are defined using pure UML class diagrams and the static semantics are specified with constraint using the Object Constraint Language (OCL). The specification of presentation/visualization information necessitated extensions to UML, mainly in the form of predefined object attributes for such things as icon file names, colors, line types etc. These could be specified using UML attributes. However, a design decision was made that, since the visualization information only pertains to GME 2000 and using GME 2000 features would make the environment more user-friendly, extensions to UML were justified.

It is important to examine the use of constraints in defining a modeling language. Some semantic rules cannot be visually specified using UML or the extended UML. These rules require the use of textual (OCL) constraints. However, the constraints can be parsed and evaluated during the construction of models. GME 2000 ensures that the constraints are met by verifying that the model does not violate any constraints defined for the paradigm. The tool designer can even specify when to check certain constraints and whether or not a constraint can be overridden. Some models may need to *temporarily* violate a constraint. For example, if the constraint says that every Process must be connected to at least one Conveyer, and every Conveyer must be connected to at least two Processes, how do you begin construction of a new model? You must allow the user to temporarily violate the constraint so they can complete the model. However, all constraints should be verified before model interpreter occurs.

Another feature that metamodeling allows is the evolution of the system over time. In Figure 1, two types

of evolution are shown: application evolution and environment evolution. For application evolution, the MIC environment must support the ability to add new or modify existing modeling interpreters to compensate for changing application requirements. For example, if the run time system changes from a Unix system to a Windows platform, some changes to the generated system may be required. In this case, changes to the modeling language are not needed.

For environment evolution, the system needs the ability to modify the modeling environment as the system requirements change over time. This could be due to a new analysis tool that requires information that cannot be captured in the current modeling language or to improve the expressivity to the language. With MIC, the metamodel can be modified to improve the domain specific language and then a new configuration for GME 2000 can be generated. One important aspect of environment evolution is the problem of *model migration*. Models need to be migrated to the improved paradigm to eliminate the need to reconstruct them manually. For some cases, the GME 2000 tools support model migration. In the general case, research is ongoing as to how to perform this translation process.

For more information a detailed description of the metamodeling environment can be found in [4].

4. Verification of Domain Specific Models

An added benefit to using MIC is the ability to perform some verification and validation at the model level. At this higher level of abstraction, the user can concentrate on the models and their intended meaning instead of trying to decipher source code to determine whether some problem was an implementation or design flaw. Additionally, the user can perform diagnostics as to why a verification routine failed rather than employing implementers to check the validity of their source code.

As usual, this process is done through the use of a model interpreter. Model interpreters can be provided that perform verification or that provide detailed information from the models to outside verification tools. Since the models should capture all of the information necessary to analyze and synthesize the system, they will also capture all of the information necessary to perform verification of the system models. In many cases, it is cheaper, easier, and quicker to perform the verification at the model level instead of at the system code level. By using the MIC interpreter interfaces, the system developers are able to "attach" model verification routines to the modeling tools instead of trying to verify the artifacts of the system generation process.

One example of model verification using MIC is a project where the modeling language allowed for the behavioral modeling of high assurance systems [7]. These models were converted into Ordered Binary Decision Diagrams (OBDDs) [8] and then symbolically evaluated. This symbolic search through the models allowed extremely large modeled behavioral spaces to be examined. The result of the analysis was a set of reliability and safety data derived from the models. The system users dealt with behavior models and reliability and safety data, which was a natural form for the users. The system users were shielded from the details of the verification and model checking routines.

A similar technique was used in to verify that selected models would meet run time performance constraints [9, 10] by checking the constraints against the models. By assuring that only valid models would be evaluated, the user did not have to deal with interpreting and evaluating models that did not meet the constraints.

5. An Example MIC Application

Now, lets take a quick look at an example application to show the utility of MIC in practice. MILAN is a model-based, extensible simulation framework that facilitates rapid evaluation of different performance metrics, such as power, latency, and throughput, at multiple levels of granularity for a large set of embedded systems by seamlessly integrating different, widely-used simulators into a unified environment. The MILAN framework is aimed at the design of embedded high-performance computing platforms, of System-on-Chip (SoC) architectures for embedded systems, and for the hardware/software co-design of heterogeneous systems. MILAN is a multi-year effort; only preliminary results and future plans are discussed in this paper. MILAN is constructed using the MIC technology and GME 2000 [10].

Figure 3 shows the architecture of the MILAN framework. At the top is the Generic Modeling Environment configured to support the modeling language developed specifically for MILAN. There are three kinds of models in MILAN: resource models, application models and explicit constraints. The application models are based on a hierarchical signal flow representation with important extensions. Most notably, the modeling language allows for the specification of explicit design or implementation alternatives of any component. Among the other features are the ability to model synchronous and asynchronous dataflows and the ability to mix synchronous and asynchronous dataflows. At the lowest levels in the model hierarchy, the user must specify the function to be executed. The model interpreters, denoted by the circles containing *I*s, can take care of generating the "glue code" necessary to execute the system as well as any scheduling that needs to occur.

The modeling of alternatives allows the entire design space of the application to be captured as opposed to a point solution. To manage this design space, application requirements, resource constraints and other specifications are captured as explicit constraints in the models. The resource models capture the available hardware components and their interconnectivity.

The Design-Space Exploration and Pruning tool takes the potentially very large design space and applies the constraints using a symbolic constraint satisfaction technique to find the set of solutions that satisfy all the constraints. The modeling methodology and the design-space exploration technique are described in detail in subsequent sections. The goal of design space exploration is to identify a small number of valid candidate designs. To find the balance between an under-constrained and an over-constrained model is a highly iterative, human-in-the-loop process. One of the design goals of the modeling environment and the design-space exploration tools is to support the automation of this activity.

The next step in the design process is to utilize the integrated simulators to simulate the candidate designs one-by-one. Each supported simulator has a corresponding model interpreter that configures the simulator from the system models.

MILAN supports several different classes of simulators. Functional simulators, such as Matlab or SystemC, verify the functionality of the application without regards to performance or power. The integrated high-level simulator provides a rapid, reasonably accurate estimate of different performance criteria of the system. Lower-level power and performance simulators, such as SimpleScalar or SimplePower, are also supported. While they can be very accurate, their slow speed may prevent the simulation of the whole system.

One of the major challenges of an integrated simulation framework like MILAN is how to interpret the results of dissimilar simulators. In our architecture, model interpreters specific to each individual simulator take results and feed them back to the models. Results from a SimpleScalar simulation of a component can be stored in the models in the form of performance attributes that the high-level simulator can utilize in evaluating the performance of the whole system. This allows the different levels of simulation to interact through the models. Note that the interpretation of the results can be a human-in-the-loop process. For example, we do not plan any automatic model modification based on a functional simulation in Matlab.

Once a candidate design has been selected, through the process of simulation and design space exploration, the target application can be automatically synthesized. This step is fairly similar to driving the simulators. Instead of the semantics of the target simulator, the semantics of the runtime system have to be observed by the interpreter.

The MILAN Modeling Language

A preliminary representation method has been selected that partitions the system into three distinct classes of models: application models, resource models and constraints. Application models describe the task to be performed while the resource models describe the physical hardware available. Constraints specify requirements. A mapping between components of the application and available resource models is used to capture the space of possible design choices limited by the design constraints.

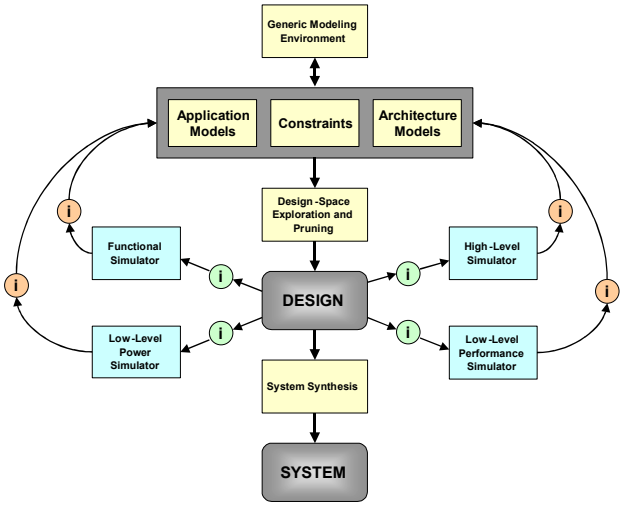

Figure 3: The MILAN Architecture

Application models are currently constructed using asynchronous and synchronous dataflow diagrams. These models are strongly typed. Mixed models are supported – those that have synchronous and asynchronous components. The models can include explicit implementation alternatives, which represent the application design space. At the leaf levels in the hierarchy, the user must provide the implementations of the data flow blocks.

Design constraints capture system requirements such as timing, performance, power, cost, etc. Moreover, resource constraints and other information also need to be specified as part of the models. In MILAN, the Object Constraint Language (OCL) is used to specify constraints in a formal manner.

Design Space Exploration

The approach we have chosen for rapid exploration of large design spaces relies on a symbolic representation based on Ordered Binary Decision Diagrams (OBDD). In this symbolic representation, a set (or space) is represented mathematically by its characteristic function. Constraints express complex relationships and bounds over composite properties of elements. Constraint application is a logical conjunction of the functions derived from the models and the constraints. The resultant Boolean function represents the pruned design space. The principal advantage of this approach is that constraint satisfaction is accomplished without the enumeration of the entire space. Eliminating the need for enumeration makes the approach highly scalable, and particularly attractive to representing extremely large design spaces.

Currently Integrated Simulators

Several simulation engines have already been implemented in MILAN. Matlab can be used for functional verification of the application models. SimpleScalar can also be configured from the application models. Hardware models are used to generate SystemC simulations. A system level estimator has been integrated. Minor modifications allowed the SimpleScalar interpreter to support the use of PowerAnalyzer, a power aware cycle accurate simulator. Future simulation engines to be integrated include a VHDL simulator and other power-aware processor simulators.

Work is underway to solve the problem of automatically updating model information based on simulation results. The system level simulator should utilize results gained from component simulators. This "vertical simulation integration" allows for increased clarity in the system level simulation results. "Feedback interpreter generation" is part of our ongoing work in this area.

All of these interpreters make use of the high level interpreter interface provided by GME 2000. One of the primary advantages of using MIC is that several simulators can be configured from the same set of models. In effect, a single system specification is reused in providing the differing simulators their inputs. This is possible due to invoking multiple interpreters on a single model.

6. Conclusions

MIC is a proven technology for developing and evolving complex computer based systems. Using MIC allows for the creation of graphical models that define the syntax, semantics, and presentation of a domain specific language. These language specifications can be automatically verified for legality and then used to configure a new domain specific tool environment. GME 2000 is a cornerstone of these domain specific design tools. A set of interfaces allows for access to the modeled information. This information can then be used to verify and validated the models as well as for generation and configuration of the run time system. One of the advantages of using MIC is the ability for the end user to design in a natural domain instead of directly writing source code. This allows the users to work with their design environments at a higher level of abstraction while ensuring actual run time systems can be created from the higher level models.

The MILAN framework is an excellent example of a MIC system. While it is currently being developed, enough data is available to show the utility of the project. It has been used to demonstrate the design and simulation of several real-world problems. MILAN is only intended to show the utility of the MIC technology. Model Integrated Computing (MIC) will allow the system to *evolve* with the ever-changing simulation requirements of embedded computing applications.

7. Acknowledgements

I would like to thank DARPA for their support for a portion of the work presented in this paper. MILAN is funded by DARPA under contract number F33615-C-00-1633, which is monitored by Wright Patterson Air Force Base.

8. References

[1] Sztipanovits, J. and Karsai, G.: "Model-Integrated Computing," IEEE Computer, April, 1997.

[2] Ledeczi, A. et.al.: "Metaprogrammable Toolkit for Model-Integrated Computing," Engineering of Computer Based Systems (ECBS), Nashville, TN, March, 1999.

[3] Ledeczi A., et.al.: "The Generic Modeling Environment", Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001.

[4] GME 2000 User's Manual, available from http://www.isis.vanderbilt.edu.

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

[6] Nordstrom G.: "Formalizing the Specification of Graphical Modeling Languages", Proceedings of the IEEE Aerospace 2000 Conference, CD-ROM Reference 10.0402, Big Sky, MT, March, 2000.

[7] Davis J., Scott J., Sztipanovits J., Martinez M.: "Multi-Domain Surety Modeling and Analysis for High Assurance Systems", Proceedings of the Engineering of Computer Based Systems, pp. 254-260, Nashville, TN , March, 1999.

[8] Bryant, R.E., "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams", Technical Report CMU-CS-92-160, School of Computer Science, Carnegie Mellon University, June 1992.

[9] Bapty T., Neema S., Scott J., Sztipanovits J., Asaad S.: "Model-Integrated Tools for the Design of Dynamically Reconfigurable Systems", VLSI Design, 10, 3, pp. 281-306, 2000.

[10] Agrawal A. et. al.: "MILAN: A Model Based Integrated Simulation Framework for Design of Embedded Systems", Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2001), Snowbird, Utah, June 2001.