Composing Domain-Specific Design Environments

Model-integrated computing can help compose domain-specific design environments rapidly and cost-effectively. The authors discuss the toolset that implements MIC and present a practical application of the technology—a tool environment for the process industry.

Ákos Lédeczi
Árpád Bakay
Miklós Maróti
Péter Völgyesi
Greg Nordstrom
Jonathan Sprinkle
Gábor Karsai
Institute for Software Integrated Systems, Vanderbilt University

What do Rational Rose, Simulink, and LabVIEW have in common? At first, these tools seem very different. Rational Rose (http://www.rational.com) is a visual modeling tool, Simulink (http://www.mathworks.com) is a hierarchical block-diagram design and simulation tool, and LabVIEW (http://www.ni.com) is a graphical programming development environment. Despite the different terminology, these three tools share a common underlying theme: Each is an integrated set of modeling, model analysis, simulation, and code-generation tools that help design and implement computer-based systems (CBSs) in a specific, well-defined engineering field.

These tools and other popular domain-specific integrated development environments can help capture specifications in the form of domain models. They also support the design process by automating analysis and simulating essential system behavior. In addition, they can automatically generate, configure, and integrate target application components. These environments translate the verified design—expressed in a domain-specific, primarily graphical modeling formalism—into a variety of artifacts that constitute a CBS implementation. These artifacts can include glue code, database schema, and configuration tables.

These tools use domain-specific modeling languages that allow developers to represent essential design views and to both formally express and automatically enforce integrity constraints. These tools also support model composition that is synergistic with the design process in the particular engineering domain. Other benefits include having integrated models as opposed to relying merely on source code. In addition, the common input—that is, the shared design model—guarantees the consistency of different analysis results as long as all of the applied generators are correct.

While the industry understands the well-documented benefits of domain-specific, integrated modeling, analysis, and application-generation environments, their high cost represents a significant block to wide acceptance and application. Consequently, these tools are available only for domains with large markets in which high volume offsets the substantial initial investment cost. For CBSs in smaller, specialized domains, or even for single projects, the industry needs technology that can help rapidly and efficiently compose these environments from reusable components.

REUSABLE FRAMEWORK

Much of our research at the Institute for Software Integrated Systems at Vanderbilt University focuses on the rapid and cost-effective composition of domain-specific design environments. One result of this research is *model-integrated computing,* a model-oriented technology.[1] MIC uses metamodeling to define the domain modeling language and model integrity constraints and uses these metamodels to automatically compose a domain-specific design environment. System designers use the resulting environment to create domain models to analyze and automatically translate into the target CBS implementation.

Metamodeling

To be useful, a reusable framework for creating domain-specific design environments must support a

0018-9162/01/$17.00 © 2001 IEEE

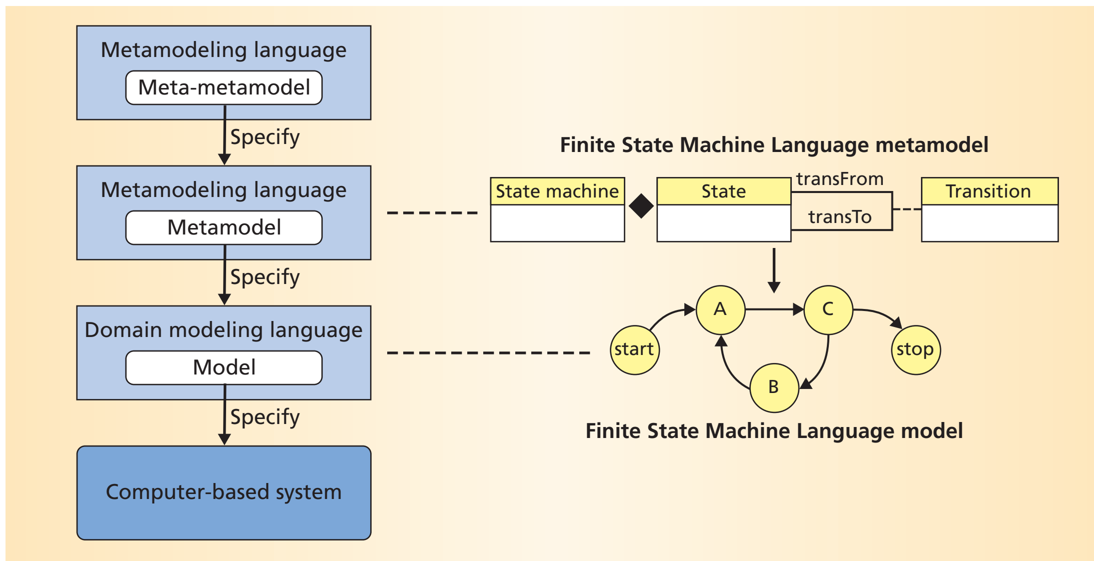

Figure 1. Model-integrated computing (MIC) four-layer metamodeling architecture. The state machine UML class on the right side is a container that can hold states. Transitions are associations between states.

set of abstract modeling concepts that are generic enough to be applicable to a wide range of domains. These concepts might include containment, module interconnection, multiaspect modeling, inheritance, and textual-numerical attributes. The framework instantiates customized concepts for each target domain, possibly multiple times, to support domain concepts directly.

The MIC framework consistently applies a metalevel architecture. As Figure 1 shows, MIC follows the standard four-layer metamodeling architecture applied in the Unified Modeling Language specification.[2] The one predefined language in this scheme—the metamodeling language—is rich enough to describe modeling languages for a wide variety of domains. As a consequence, it can describe itself in the form of a meta-metamodel. A metamodel specifies a domain modeling language that, in turn, specifies CBS models in the particular domain. The key to this four-layer architecture is that a layer is always described in terms of the next higher layer in the hierarchy.

Static semantics

The metamodels specify the domain modeling language, or, more precisely, its syntax. They do not entirely specify the language's static semantics—the set of rules that specify the well-formedness of domain models. UML class diagrams allow the specification of some basic rules—for example, the multiplicity of associations. For more complex specifications, UML includes the Object Constraint Language (OCL),[3] a textual predicate logic language. MIC adopts OCL as well. Metamodels consist of UML class diagrams and OCL constraints.

Suppose the finite state machines in the target domain shown in Figure 1 must not allow state transitions from a state to itself. A UML class diagram alone cannot specify such a rule. Rather, the following OCL expression must be attached to states:

```
self.transTo->forAll(s | s <> self)
```

where `self` and `forAll` are OCL keywords, while `transTo` is a role name of the transition association. Checking these constraints programmatically is relatively straightforward. The program simply needs to evaluate the expressions in the context of every applicable model object.

Dynamic semantics

MIC's strength rests in its dual use of domain-specific models. On one hand, you can verify the models against different domain-specific criteria because a comprehensive set of analysis tools is typically available in any mature engineering field. MIC enables the automatic configuration of these tools with information captured in the domain models. For example, you can use this approach to carry out a schedulability analysis of a real-time system or a diagnosability analysis of an aircraft subsystem.

On the other hand, MIC automatically translates the domain models into the actual CBS implementation or simulation. In each domain, there is at least one execution platform, such as a real-time operating system, a Java virtual machine, or a simulator package. Each of these execution platforms has its own executable modeling language with clearly defined execution semantics.

First, a translator transforms the domain models into executable models—for example, in the form of API calls to a real-time OS. This mapping process

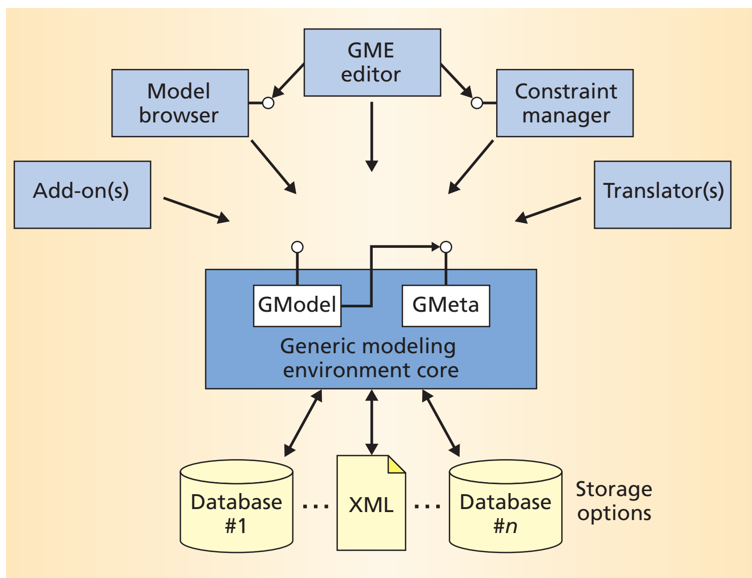

Figure 2. Generic modeling environment architecture. GME supports different storage formats ranging from relational databases to proprietary binary file formats to XML. GModel and GMeta are two key GME components. GModel implements the modeling concepts for the given paradigm, while GMeta defines the modeling paradigm. Both components expose their services through COM interfaces. The user interacts with the components at the top of the architecture.

assigns dynamic semantics to the models. Next, the execution platform executes those models. A transformation engine and an execution engine together realize the domain semantics.

Translators

Clearly, translators are key MIC components. Traditionally, developers implement translators manually in standard languages like C++. A translator maps the domain modeling language to the executable modeling language. If formal models of the mapping—along with its inputs and outputs—are available, a meta-level translator can generate the translator code.

The input model is already available as the metamodel that specifies the domain language. The executable modeling language metamodels can capture the output models in a similar manner. Capturing these metamodels usually involves some reverse engineering because execution platforms typically are not formally documented. The most challenging problem seems to be to model the mapping itself and to create a metatranslator. We are actively researching these issues.[4]

MIC process

Typically, different classes of users are associated with MIC technology. The process of developing a domain-specific environment and applying it in the given domain usually works in the following way: Metamodelers define the domain modeling language, use the metamodeling environment to formally specify the language, and generate the environment automatically. The process is highly iterative. Next, domain modelers use the new domain-specific environment to build models of the CBS they are developing. They then apply the translators to analyze the design and generate the application that the vendor eventually ships to users.

GENERIC MODELING ENVIRONMENT

The *generic modeling environment* (http://www.isis.vanderbilt.edu/search.asp?CMD=SEARCH), a tool suite that supports MIC, has an open, extensible, and modular component-based architecture. GME's main elements include a metaprogrammable graphical editor, a metamodeling environment, and an integrated constraint manager. The tool suite supports generic database connectivity and provides various interfaces for programmatic access to model data. Domain language specifications configure the metaprogrammable graphical editor to define how the architecture will map domain-specific idioms to the general concepts the GME supports.

GME architecture

GME uses a modular component architecture like the one shown in Figure 2. The two key components of GME—GModel and GMeta—are metaprogrammable: GModel uses the GMeta services for self-configuration, while GMeta configures itself by reading the metaspecifications. Each of these components exposes its services through a set of public interfaces. GME's architecture is based on Microsoft's COM technology.

The GModel component exposes a set of events such as "object deleted" and "attribute changed." External components can register to receive some or all of these events. GModel automatically invokes these components when the events occur. Add-ons—event-based components—are useful for extending GME user interface capabilities or for executing domain-specific operations.

The constraint manager evaluates the constraint expressions to verify the models against the OCL constraints the metamodels contain. GME can attach constraints to events. While the user can explicitly invoke the constraint manager, these event-driven constraints also trigger it. The GME editor component has no special privileges in this architecture. All other components have the same access rights and use the same set of GMeta and GModel COM interfaces to the GME. Any operation that can be accomplished through the editor can also be done programmatically through the interfaces. This flexible and extensible architecture allows the seamless integration of existing tools and services that a given target domain needs to support.

Metamodeling environment

You can use the same set of GME tools for metamodeling and domain modeling. The metamodeling environment is a domain-specific integrated design environment, and the metamodeling language is just another domain language. A metamodel translator generates the specifications that configure the GME for a given domain. The same translator generates the metamodeling environment configuration when it translates the meta-metamodels.

The reusability of metamodels from domain to domain is as important as the reusability of domain models from application to application. Ideally, the metamodeler uses a library of metamodels of important subdomains to extend and compose them to specify new domain languages. These subdomains might include signal-flow variations, finite state machines, data type specifications, fault propagation graphs, or petri nets. The extension and composition mechanisms should not modify the original metamodels, just as subclasses do not modify baseclasses in OO programming. Changes in the metamodel libraries, reflecting a better understanding of the given subdomain, propagate to the metamodels that use them.

By precisely specifying the extension and composition rules, a translator can automatically migrate models specified in the original domain language to comply with the new, extended and composed modeling language.[5]

Type hierarchy

To facilitate model reuse and maintenance, GME supports model types, instances, and type inheritance, which closely resemble OO language concepts. The only significant difference is that GME model types are similar in appearance to model instances, which are graphical, have attributes, and contain parts. By default, a model created from scratch is a type. A subtype or an instance of a model type depends on the type. Any modification of parts propagates down the inheritance hierarchy. For example, if a part is deleted in a type, the same part will be automatically deleted in all of its instances, subtypes, and instances of subtypes all the way down the inheritance hierarchy. A set of well-defined rules specifies the exact behavior of type inheritance in GME.

Data access and extensibility

Because data and tool integration is one of GME's primary application areas, its design goals include flexible data access and standards-compliant extensibility. GME is completely component-based, and its components include public interfaces. Most notably, the GME editor (the visualization component), the model storage and logic, and the metamodeling module are separated by interfaces that are accessible to user-written components.

Because GME uses COM, the primary languages for integration are C++ and Visual Basic, but Java, Python, and other languages are also viable. Access is bidirectional and fully transactional, which makes different online modeling scenarios feasible. For example, you can use the GME editor itself as the user interface of a generated application to provide feedback about models in the proper context. Bidirectional access makes it possible to convert legacy data into models automatically.

Programming GME at the component level requires COM programming expertise. Several methods provide easier access to the component level through simpler interfaces—albeit with limited functionality. The GME pattern-based report language provides reporting capabilities by interpreting macro definitions in a simple text input file. GME also provides an easy-to-use, extensible C++ API layered on top of the COM interfaces. In addition, GME offers bidirectional XML access for both model and metamodel information.

Model visualization can also be customized within the GME editor. A separate software component is responsible for drawing the models and handling user actions. This component can be replaced on a paradigm-per-paradigm or even on a model-by-model basis to provide highly domain-specific visualization of the models.

GME's flexible and extensible modular component architecture allows the seamless integration of existing tools and services that a given target domain needs to support.

ACTIVITY MODELING TOOL

The activity modeling tool environment, a representative yet relatively simple application of MIC in general, and GME in particular, helps build custom process monitoring and simulation applications for chemical plants. AMT provides a means to model all the components necessary to interface to the real-time database, define custom data processing, create an operator interface, and interface to a COTS process simulator. The translators use a set of integrated models to synthesize the custom process monitoring and simulation application.

A remotely running process monitoring and control database provides real-time data input to the target systems. The Aspen Plus Steady State (http://www.aspentech.com) process simulation engine uses a COM interface to configure and execute simulations. The Aspen simulation engine's location can range from a local PC to a remote cluster of supercomputers.

The generated application is based on a client-server architecture. The client is a remotely configurable custom-built Java applet that serves as the operator interface. The models specify the configuration information that defines what user interface elements to display, how to lay them out, and how to connect them to data

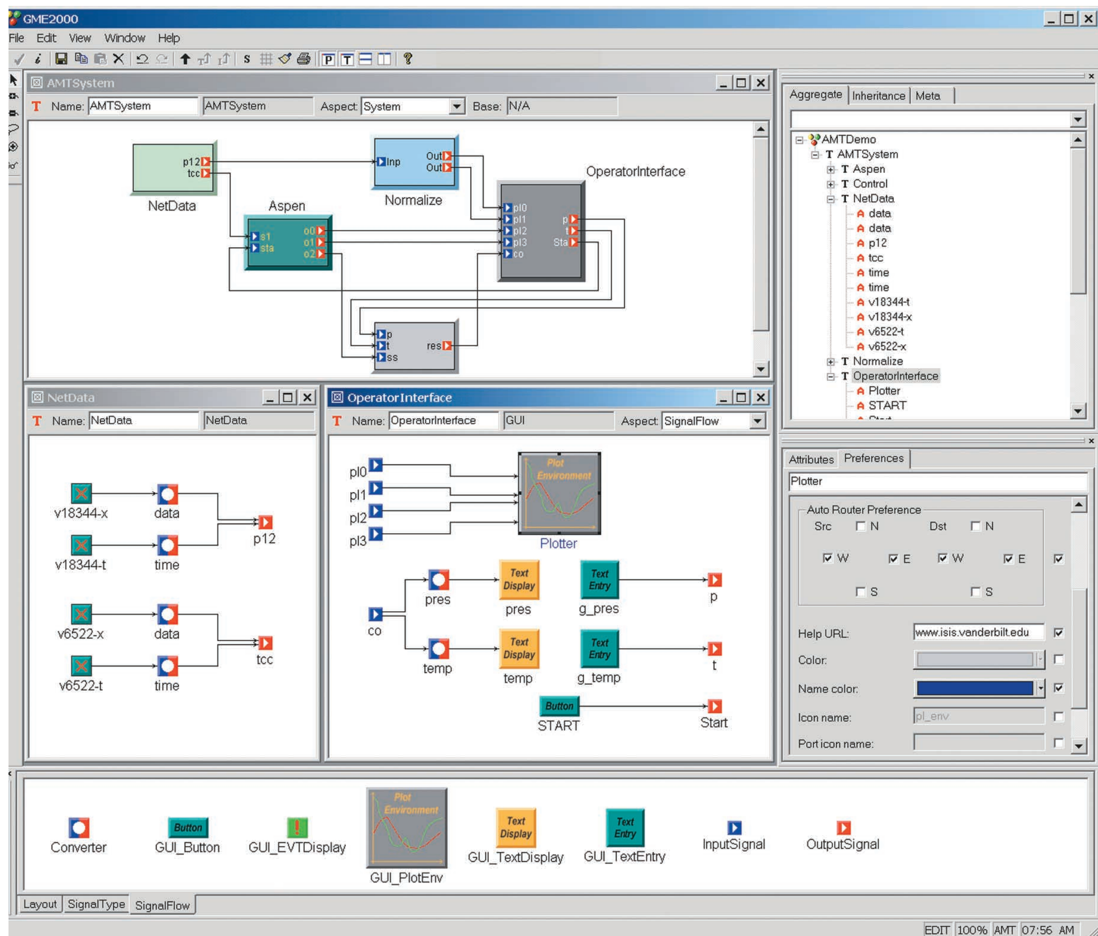

Figure 3. Activity Modeling Tool system. The top-level application model contains a variable database, a simulator, and an operator interface model along with custom processing blocks that contain hierarchical signal flow graphs. The operator or interface model illustrates aspect modeling.

sources. The server runs on top of a dynamically scheduled dataflow kernel. Some dataflow nodes interface with the variable database, the simulator, and the operator interface. Other dataflow nodes implement the custom processing steps. The translator configures the interfaces and generates the dataflow components. The user only needs to provide the custom processing primitives in the form of C functions.

The domain modeling language is based on a strongly typed, hierarchical signal-flow paradigm. The language can specify both simple and composite data types and associate them with signals. A set of constraints enforces type consistency along signal connections.

Extensibility

AMT's use of time-stamped data values illustrates GME's extensibility. A composite data type model specifies two fields: data and time. However, the database provides the data and its time stamp as two separate variables. Consequently, as the NetData model in Figure 3 shows, the interface models typically contain pairs of variable atoms connected to field specifiers connected to signal atoms.

A simple Visual Basic add-on triggered by the creation of a new variable atom automates the creation of these repetitive model structures. When the user selects one of the available variables, this add-on invokes the NetData browser, which is part of the database interface package toolset. The add-on then creates the corresponding atoms for the time stamp, field specifiers, and output signal; sets their attributes; and connects them automatically. This reduces modeling time and decreases the chance of modeling errors.

Multiaspect modeling

Figure 3 also shows an operator interface model that illustrates the use of aspects. In the SignalFlow aspect, the GUI components are wired to signals. The SignalType aspect uses references to data types defined elsewhere in the models to specify the data types and to associate them with the signals. The Layout aspect specifies how to position the operator interface GUI components on the screen. This aspect only needs to show the GUI widgets and a set of tile atoms whose number and relative position determine the final layout. The signal flow, data type specifications, and GUI layout are largely independent

concerns. Separating them in different aspects improves the readability of the models.

The unique needs of this particular domain—specifically, interfacing with existing heterogeneous components—precluded the use of commercially available environments. The complexity of the requirements made developing a custom-made toolset cost-prohibitive. Using the GME, on the other hand, only required an eight-person-month effort, including requirements specification, design, implementation, and testing.

RELATED RESEARCH

Unlike MIC, which targets system-level modeling, the majority of the research on domain-specific languages[6] and visual languages[7] specifically focuses on programming. Among the research groups working on configurable visual editors, some target only visual programming languages.[8,9] Many only address the visualization aspect of the problem—the domain-specific tools they generate are diagram editors.[8-10] In contrast, integrated toolsets, such as the GME, bring substantially more domain knowledge to bear during the system design process. This is similar to the difference between a text editor with syntax highlighting, such as Emacs, and a full-featured integrated development environment, such as Borland JBuilder or Microsoft Visual Studio.

The GME also contains a diagram editor, but it is only an optional component of the entire tool environment. The metamodels precisely describe a domain-specific language consisting of entities, relationships, and attributes configuring the persistence engine and the GME Core, which is similar to an object store. Additionally, the metamodels contain information regarding the visual representation and manipulation of the graphical idioms the diagram editor manages—for example, specifying that a connection depicts a certain kind of relationship. However, these are only hints that the GME editor follows by default, but other user interfaces can override them. For example, we are working on a spreadsheet-like table editor for GME to provide a more natural interface for certain kinds of modeling domains. Even the XML persistence format can be considered a textual—although not too user-friendly—interface to the models.

In this sense, our approach is similar to Microsoft's Intentional Programming model,[11] which stores the user's "intentions" as interrelated objects that can be visualized in multiple ways. Each kind of intention is associated with its own translator that generates traditional code. GME does not automatically support such a fine-grained translation process, but the internal structure of GME translators reflects a similar architecture.

CONFIGURABLE ENVIRONMENTS

GME offers a metaprogrammable modeling environment, arguably the first of a new generation of configurable tools. Many environments available today are configurable but not metaprogrammable. GME's most important distinguishing characteristic is the consistent application of a metalevel architecture enforcing the strict relationship between metamodels, models, and the generated systems, guaranteeing consistency under all circumstances during the target system's entire lifetime.

Two well-known configurable tool environments clearly stand above the rest: Dome (http://www.src.honeywell.com/dome/) from Honeywell Laboratories, and MetaEdit+ (http://www.metacase.com), a commercial environment from MetaCase Consulting of Finland.

Dome models are basically a collection of linked diagrams. Alter, a powerful LISP-based language, provides bidirectional programmatic access to the models. Dome configures the toolset, but its language specifications only control the creation of new model elements, not their subsequent life. While such an approach enables the modification of the domain-modeling environment while it is running, arguably an elegant feature, it does not maintain consistency in the domain models.

The MetaEdit+ toolset's features include data storage in an object-oriented database; multiuser capabilities with access and transaction control; a symbol editor for defining visualization; and an elegant built-in language for generating program source code, reports, and even formatted documentation from the models. However, MetaEdit+ has a closed architecture that does not offer bidirectional access to the model data, resulting in a nonextensible environment. Even read-only access is only available through a proprietary scripting language. Furthermore, developers do not configure a MetaEdit+ environment through metamodels, but through a series of dialog boxes similar to application-generator wizards.

Both Dome and MetaEdit+ offer a limited and fixed set of parameterizable constraint types. A unique feature of GME is that it uses OCL as its constraint language, which makes it more adaptable to different domains. Practice has shown that powerful constraints are absolutely necessary to maintain consistency in real-world models—models that are magnitudes larger and more complex than the typical demonstration applications. Neither Dome nor MetaEdit+ has built-in type inheritance support, an indispensable technique for model reuse and maintenance.

Powerful constraints are absolutely necessary to maintain consistency in real-world models—models that are magnitudes larger and more complex than the typical demonstration applications.

INDUSTRIAL APPLICATIONS

As the "MIC Applications" sidebar indicates, industry uses GME extensively in a wide range of application domains for modeling system structure

(computer hardware and software components), failure modes and fault-tolerant behavior, different kinds of structured data (like documents and courseware), and even metamodels. Modeling computations and using the models for code synthesis is one of its most attractive application areas.

Various research groups have developed environments exclusively targeting embedded software modeling formalisms. Notable examples are Moses by ETH Zurich (http://www.tik.ee.ethz.ch/~moses/) and Ptolemy by the University of California, Berkeley (http://ptolemy.eecs.berkeley.edu). Because they are not general-purpose metaprogrammable modeling environments, they go further in their specialized area by including an integrated execution or simulation engine. Consequently, it is possible to model simple systems and run simulations in a short time, making the tools excellent for academic demonstrations, prototyping, and proof-of-concept simulations.

Both Ptolemy and Moses include several prebuilt domains or "models of computation."[12] These domains allow the free composition of formalisms, although some such heterogeneous models may prove to be semantically incorrect. In GME, on the other hand, the metamodels explicitly capture and control paradigm composition.

The high cost of developing domain-specific, integrated modeling, analysis, and application-generation environments prevents their penetration into narrower engineering fields that have limited user bases. MIC and its toolset provide a way to compose

MIC Applications

Developers and researchers have applied model-integrated computing in general and GME or its predecessors[1] in particular to several real-world applications. The Saturn Site Production Flow (SSPF) system monitors Saturn's automotive manufacturing process, providing key production measures to managers in real time.[2] The system models describe the manufacturing processes down to the machine level, the buffers between the processes (such as the conveyor belts), the instrumentation, and how the information is presented to the user. The translators generate various 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 using the system, Saturn's Spring Hill, Tennessee, site reported significant gains in productivity, citing the SSPF technology as a key factor in the increase. A second Saturn installation in Delaware took two weeks of modeling time and only two hours of installation time. No code modification was required, which illustrates MIC's power.

The primary objective of the State Space Analysis Tool (SSAT) developed for Sandia National Laboratories is to increase the reliability, safety, and security of high-impact systems by analyzing design models.[3] Integrated system models built using GME can automatically generate diagnostic information for analysis. The specific analysis techniques include verification of requirements consistency, validation of design models versus requirements models, simulation of system behavior (including forward and backward system execution), safety and reliability analysis (using existing proprietary tools), and automatic fault-tree generation.

Developers have also used MIC to build tools for fault detection, isolation, and recovery applications. The models capture the component hierarchy of a complex physical system—a next-generation aircraft in one case[4] and the International Space Station in another.[5] The models also specify the anticipated failure modes of the components and how failures propagate through the system, inducing various functional discrepancies. A real-time fault diagnostics system uses these models to isolate fault sources using observations of sensor data. Another use of the models is design-time diagnosability analysis to determine sensor coverage, size of ambiguity groups for various fault scenarios, timeliness of diagnosis results in the onboard system, and other relevant domain-specific metrics.

These example systems illustrate a major characteristic of MIC that is often overlooked: MIC is not about software modeling and generation. The domain-specific languages it supports are not visual programming languages. Instead, MIC is a technology that supports system development and evolution. Software is an important part of computer-based systems, but it is not the only part. In some cases, such as the AMT project, parts of the system models correspond to software entities; in other cases, however, software does not even appear in the models directly.

References

1. G. Karsai, "A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming," *Computer*, Mar. 1995, pp. 36-44.
2. E. Long, A. Misra, and J. Sztipanovits, "Increasing Productivity at Saturn," *Computer*, Aug. 1998, pp. 35-43.
3. J. Davis et al., "Multi-Domain Surety Modeling and Analysis for High Assurance Systems," *Proc. Eng. Computer-Based Systems* (ECBS 99), IEEE Press, Piscataway, N.J., Mar. 1999, pp. 254-260.
4. L. Atlas et al., "An Evolvable Tri-Reasoner IVHM System," *Proc. 2001 IEEE Aerospace Conference*, IEEE Press, Piscataway, N.J., 2001, pp. 3023-3037.
5. J.R. Carnes, A. Misra, and J. Sztipanovits, "Model-Integrated Toolset for Fault Detection, Isolation and Recovery (FDIR)," *Proc. IEEE Symp. and Workshop Eng. of Computer-Based Systems*, IEEE Press, Piscataway, N.J., 1996, pp. 356-363.

such environments cost-effectively and rapidly by using a metalevel architecture to specify the domain-specific modeling language and integrity constraints. This process can significantly reduce a CBS's development time and costs. ✸

Acknowledgments

We thank DARPA ITO, the Boeing Company, and the DuPont Old Hickory Plant for their generous sponsorship of our research. We also thank Frank DeCaria, who provided invaluable domain knowledge, and Jason Garrett, who was the primary implementer in the AMT project.

References

1. J. Sztipanovits and G. Karsai, "Model-Integrated Computing," *Computer*, Apr. 1997, pp. 110-112.
2. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison Wesley Longman, Reading, Mass., 1998.
3. J.B. Warmer and A.G. Kleppe, *The Object Constraint Language: Precise Modeling With UML*, Addison Wesley Longman, Reading, Mass., 1999.
4. G. Karsai, "Structured Specification of Model Interpreters," *Proc. Conf. Eng. of Computer-Based Systems* (ECBS 99), IEEE Press, Piscataway, N.J., Mar. 1999, pp. 84-90.
5. A. Ledeczi et al., "On Metamodel Composition," *Proc. Conf. Control Applications*, IEEE Press, Piscataway, N.J., 2001, pp. 84-90.
6. A. van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," *SIGPLAN Notices*, vol. 35, no. 6, 2000, p. 26.
7. M.M. Burnett and M.J. Baker, "A Classification System for Visual Programming Languages," *J. Visual Languages & Computing*, Sept. 1994, pp. 287-300.
8. C.A.M. Grant, "Visual Language Editing Using a Grammar-Based Visual Structure Editor," *J. Visual Languages and Computing*, Aug. 1998, pp. 351-374.
9. S.M. Uskudarli, "Generating Visual Editors for Formally Specified Languages," *Proc. IEEE Symp. Visual Languages*, IEEE Press, Piscataway, N.J., 1994, pp. 278-287.
10. O. Koth and M. Minas, "Abstractions in Graph-Transformation-Based Diagram Editors," *Electronic Notes on Theoretical Computer Science*, http://www.elsevier/inca/publications/store (current Oct. 2001).
11. C. Simonyi, "The Future Is Intentional," *Computer*, May 1999, pp. 56-57.
12. J. Davis et al., "Overview of the Ptolemy Project," http://ptolemy.eecs.berkeley.edu/publications/papers/01/overview/overview.pdf, Mar. 2001 (current Oct. 2001).

Ákos Lédeczi *is a research scientist at the Institute for Software Integrated Systems, Vanderbilt University. His current research interests include model-based synthesis and simulation of embedded systems. He received a PhD in electrical engineering from Vanderbilt University. Contact him at akos.ledeczi@vanderbilt.edu.*

Árpád Bakay *is a research scientist at the Institute for Software Integrated Systems, Vanderbilt University. His current research interests include model-integrated computing. He received a PhD in electrical engineering from the Budapest University of Technology and Economics. Contact him at arpad.bakay@vanderbilt.edu.*

Miklós Maróti *is a research assistant at the Institute for Software Integrated Systems, Vanderbilt University. His current research interests include data modeling, constraint transformations, distributed algorithms, generative programming, and active libraries. He received an MSc in mathematics from Vanderbilt University. Contact him at mmaroti@math.vanderbilt.edu.*

Péter Völgyesi *is a research instructor at the Institute for Software Integrated Systems, Vanderbilt University. His current research interests include model-integrated computing and visual programming environments. He received an MSc in technical informatics from the Budapest University of Technology and Economics. Contact him at peter.volgyesi@vanderbilt. edu.*

Greg Nordstrom *is the Intelligent Systems Group Leader at the Titan Systems Technology Division. His research interests include graphical language modeling, biotechnology, and automation modeling. He received a PhD in electrical engineering from Vanderbilt University. Contact him at greg.nordstrom@titan.com.*

Jonathan Sprinkle *is a research assistant at the Institute for Software Integrated Systems, Vanderbilt University. His research interests include model migration. He received an MSc in electrical engineering from Vanderbilt University. Contact him at jonathan.sprinkle@vanderbilt.edu.*

Gábor Karsai *is an associate professor at the Institute for Software Integrated Systems, Vanderbilt University. His research interests include design and implementation of advanced software systems for control and instrumentation, programming tools for building visual programming environments, and the theory and practice of model-integrated computing. He received a PhD in electrical engineering from Vanderbilt University. Contact him at gabor.karsai@vanderbilt.edu.*