

The Generic Modeling Environment

Technical Report

Ákos Lédeczi, Miklós Maróti and Péter Völgyesi

Institute for Software Integrated Systems, Vanderbilt University

Nashville, TN, 37221, USA

akos.ledeczi@vanderbilt.edu

ABSTRACT

This chapter introduces the concepts and techniques required for developing graphical, domain-specific modeling and program synthesis environments. It argues that a fully functional modeling environment can be quickly developed for a wide variety of engineering domains using a configurable and extensible toolset with a limited set of generic concepts. The configuration is accomplished through metamodels specifying the modeling language and methodology containing all syntactic, semantic and presentation information of the domain. The authors applied this approach to several real-world systems.

INTRODUCTION

Graphical modeling environments for system development are integrated sets of modeling, model analysis, simulation and code generation tools that aid the design of systems in a particular, well-defined engineering field. These toolsets capture specifications in the form of domain models, support the design process by automated systems analysis and simulation and automatically generate, configure or integrate components of the target applications. Examples for such domain-specific environments are Rational Rose for object oriented software development, Matlab/Simulink for signal processing and LabView for instrumentation.

Advantages of such environments are the result of their domain specificity. Domain-specific modeling methodologies enable the concise representation of essential design views, the formal expression and automated enforcement of integrity constraints and model composition that is synergistic with the design process in the domain.

While these benefits of domain-specific development environments are well understood and documented, their high cost represents a significant roadblock against their wider application. Consequently, domain specific toolsets are available commercially only for domains with large markets, where the significant initial investment is offset by high volume. For the rest of the application areas, one solution is to create configurable tools that readily provide the generic functionality of graphical development environments (creating and managing design projects, editing and combining diagrams, translating information into output formats), and let them easily be tailored to use the concepts of a given domain. Such tools can approach, albeit never fully reach, the features of an environment directly developed for a given domain. Their key advantage is that effort needed for customizing them for the domain is orders of magnitude less than a developing a custom-made toolset.

Furthermore, these tools ease the development and evaluation of new or modified modeling methodologies. As we will show, the development of a fully functional modeling environment using a configurable toolset can take from hours to days depending on the complexity of the given modeling methodology. On the other hand, the development of a custom environment from scratch is measured in man-years.

The Generic Modeling Environment (GME) (Lédeczi et al., 2001) developed at the Institute for Software Integrated Systems at Vanderbilt University (freely available at <http://www.isis.vanderbilt.edu/projects/gme>) is one of the more prominent configurable modeling environments. Its configuration is accomplished through metamodels specifying the modeling paradigm (modeling language, modeling methodology) of the application domain. The modeling paradigm contains 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.

Metamodeling is the primary method for specializing a GME instance. The metamodeling language is based on the UML class diagram notation. Metamodels also contain OCL constraints specifying the static semantics of the modeling language. These constraints are automatically enforced in the target GME instance. Additional methods for customizing GME include decorators, interpreters, and add-ons. Decorators are simple software components that can be attached to a GME instance. They are used for domain-specific visualization of the models. Interpreters and add-ons are external software components that interface with GME and provide additional domain-specific functionality including, but not limited to code generation.

The rest of this paper is organized as follows. In the next several sections the different methods for providing native support for different modeling methodologies are described in detail. Then examples are presented that illustrate these techniques. Finally, we compare two other well-known configurable modeling environments to our approach and present our conclusions.

METAMODELING

The metamodeling language is not used for defining domain models, but rather for defining domain-modeling languages. Thus, “sentences” in the meta-language define specific domain languages, while “sentences” of the domain language define specific systems. GME follows the standard four-layer modeling architecture.

The GME metamodeling methodology is implemented with GME itself. The meta-modeling language is just another domain language. The metaspecifications that configure the GME are generated by the metamodeling interpreter from the metamodels. The metamodeling language itself is generated by the same interpreter when translating the meta-metamodels.

At the metamodeling level GME provides generic modeling primitives that assist an environment designer in the specification of new modeling environments. These concepts are directly supported by the framework as stereotypes of the specific classes. Elementary types that do not contain other objects are defined as *atoms*, while *models* are composite classes. Associations between these classes are modeled using the *connection* primitive that is visualized by the modeling tool as a line between the objects. Connections can only express relationships between objects at the same hierarchy level or one level deeper with the help of ports on composite models. *References* help to overcome this limitation by enabling the user to associate objects in different model hierarchies. A reference always refers to exactly one object, which can be of any kind except connection; this establishes a relationship between the model that contains the reference and the referred object. Connections and references model relationships between at most two objects. *Sets* can be used to specify the relationship among a group of objects. Atoms, models, connections, references and sets are the first class objects (*FCO*) of the modeling framework.

The language designer can assign different attributes to first class objects. Attributes are values of predefined simple types, such as integer, string, boolean and enumeration. The meta-attribute defines the name, the value type and the default value of the attribute. The attribute value of an instantiated object is user changeable at the modeling level.

The framework provides various techniques for managing the complexity of large-scale models; the most notable concept is the introduction of aspects enabling the domain users to focus on selected parts of a design. At the metamodeling level a set of aspects are assigned to

every composite type and the visibility of each contained class can be defined for a given aspect. This powerful construct assisting the modeler in separating the concerns of multi-perspectives, is similar to views in the world of rational databases.

The modeling concepts above can be used only if the environment designer precisely defined them in the metamodel of the paradigm. In our experience, it is often necessary to associate information chunks without real semantic meaning or strict syntax to objects. This includes, for example, the specification of visualization information, such as color, style and icon for an object. Therefore we have added an extensible storage to every FCO, called the registry. The registry is a tree data structure containing the auxiliary data in the nodes of the tree. The shape and node names of the tree are not fixed in order to provide extensibility for external tools.

The metamodeling environment provides a powerful set of inheritance operators to describe specialization and to support metamodel composition. The common inheritance operator implements the semantics of the standard UML specialization, thus the specialized child type inherits all of the parent's attributes and can participate in any association the parent can participate in. Two additional operators are available to provide finer-grained control over the inheritance relation. These were specifically designed to support metamodel composition. Implementation inheritance propagates all of the parent's attributes, but only the containment association – where the parent functions as the container – to the child type. No other associations inherited in this case. Interface inheritance allows no attribute inheritance, but does allow full association inheritance, with one exception: containment relations where the parent functions as the container are not inherited. Note that the union of the two special inheritance operators gives the common inheritance, and their intersection is null.

Just as the reusability of domain models from application to application is essential, the reusability of meta-models from domain to domain is also an important consideration. In GME a library of meta-models of important sub-domains is made available to the meta-modeler, who then can pick and choose from them, extend and compose them together to specify new domain languages. The extension and composition mechanisms must not modify the original meta-models for two reasons. First, changes in the meta-model libraries, reflecting a better understanding of the given domain, for example, should propagate to the meta-models that utilize them. Second, by precisely specifying the extension and composition rules, using inheritance and equivalence operators, for instance, models specified in the original domain language can be automatically translated to comply with the new, extended and composed, modeling language. This is a simple and elegant solution to the well-known model migration problem. For more detail on metamodel composition please see (Ledeczki, Nordstrom, Karsai, Volgyesi & Maroti, 2001).

TYPES AND INSTANCES

Model reuse and tools for information maintenance between similar models is a natural requirement in large-scale models or where model composition is heavily used. The provided solutions in GME – types and instances – resemble those of object-oriented programming languages. The only significant difference is that in GME, model types are similar in appearance to model instances; they too are graphical, have attributes and contain parts.

By default, a model created from scratch – based on a meta type – is a type. A subtype or an instance of a model can be created with a simple operation, and both will depend on the type they are created from. There is one significant rule that differentiates subtypes from instances. New parts are allowed in a subtype, but not in an instance. Otherwise, contained children can be

renamed, set membership can be changed and references can be redirected in both subtypes and instances. However, objects in the containment hierarchy cannot be removed in either subtypes or instances.

The advantage of using types is clear: any modification in a type model propagates down the inheritance hierarchy. For example, if a part is deleted in a type, the same part will be automatically removed in all of its instances and subtypes – even in instances of the subtypes – all the way down the inheritance tree.

Types can contain other types as well as instances as parts. The mixture of aggregation and type inheritance introduces another kind of relationship between objects. This is best illustrated through an example. In Figure 1, there are two root type models: the *Engine* and the *Car*. The car contains an instance of an engine, *V6*, and an *ABS* type model. *V6* is an instance of the *Engine*; this relationship is indicated by the dash line. Aggregation is indicated by solid lines.

When a subtype of the *Car* is created, e.g. *Cool Car* above, we indirectly create another instance of the *Engine* (*V6*) and a subtype of the *ABS* type. This is the expected behavior as a subtype without any modification should look exactly like its base type. Notice the arrow that points from *V6* in *Cool Car* to *V6* in *Car*. Both of these are instances, but there is dependency between the two objects. If we modify *V6* in *Car*, *V6* in *Cool Car* should also be modified automatically for the same reason: if we do not modify *Cool Car* it should always look like *Car* itself. The same logic applies if we create an instance of *Cool Car* – *My Car* in Figure 1. It introduces a dependency between *V6* in *My Car* and *V6* in *Cool Car*. As the figure shows, this forms a dependency chain from *V6* in *My Car* through *V6* in *Cool car* and *V6* in *Car* all the way to the *Engine* type model.

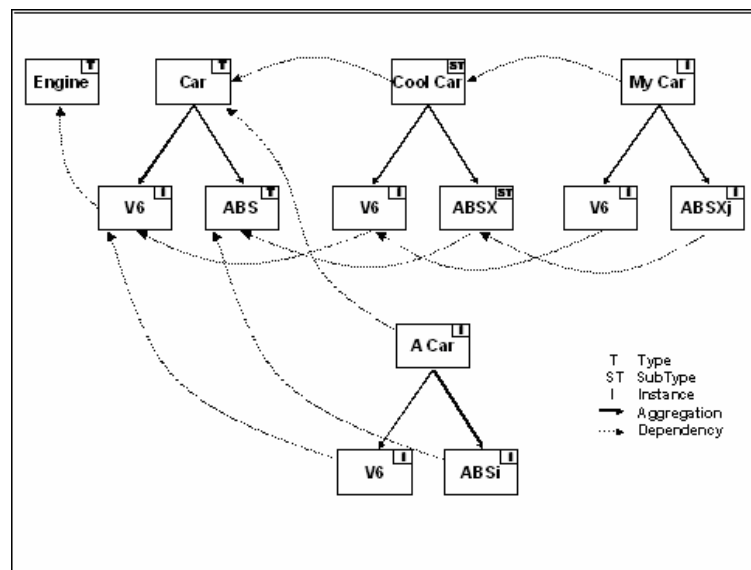


Figure 1 Model dependency chains

An interesting situation arises if we modify *V6* in *Cool Car* by changing an attribute. The question is whether an attribute change in *V6* in *Car* should propagate down to *V6* in *Cool Car* and below. Since the attribute has been overridden the dependency chain is broken up with respect to that attribute. However, if the same attribute is changed in *V6* in *Cool Car* that should

propagate down to V6 in My Car unless it has already been overridden there. Figure 2 shows the same set of models, but only from the pure type inheritance perspective.

The real strength of types and instances can be exploited with the use of model libraries. Based on predefined and verified models residing in these libraries the modeler is able to create new instances in his or her project without losing the connection to the prototype model, thus further enhancements and corrections in the original model can be easily propagated to all of its subtypes and instances automatically.

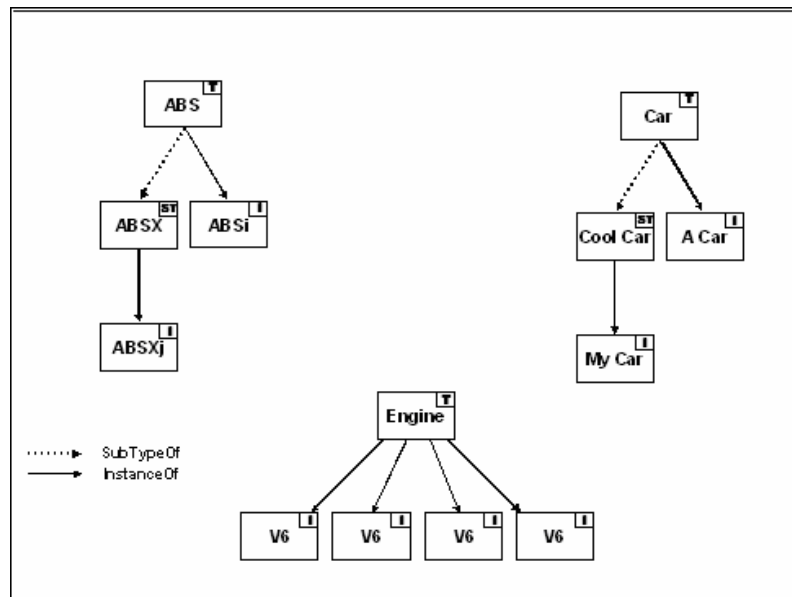


Figure 2 Type inheritance hierarchy

The type and instance feature of GME is sometimes confused with the inheritance relation in the metamodeling environment. While the meta-inheritance implements the semantic of the UML specialization—and is handled by the metainterpreter, types and instances are supported by the modeling engine, and their primary goal is to support model reuse independently of the metamodel.

CONSTRAINT MANAGEMENT

One can consider the UML class diagram-based metamodel as syntax specifications. It determines what concepts are used in the modeling language and specifies relations and attributes. It does not say much about what constitutes a correct model. We use the Object Constraint Language (OCL) for the specification of the static semantics of the modeling language. Constraints are attached to the metamodels specifying well-formedness rules.

In addition to the OCL expression, a GME constraint has a priority attached to it specifying the action the built-in constraint manager should take upon its violation. The highest priority results in an error message and the abortion of the current transaction. Lower priority violations only cause warning messages.

Constraints can be attached to editing events specifying when they must be checked. For example, the “on connect” event should be specified for a constraint that restricts the kind or

number of connections a given model can be attached to. Furthermore, all constraints can be checked on-demand at any time.

Certain pieces of information captured in the metamodel cannot be compiled directly into the paradigm configuration because of the limitations of that format. In such cases, such as the multiplicity information of containment, membership and connection cardinality definitions, the meta-interpreter automatically generates OCL constraints.

Complex and reusable constraints can be defined in constraint functions that can be called from constraints or other functions. They even support recursion. Function parameters enable the constraint developer to formalize reoccurring definitions in a generic form.

One of the most powerful facilities of the constraint system is the browser that provides an interactive window to the constraint database. The browser displays the definition, state and other attributes of each available constraint. Selected constraints can be evaluated on demand or can be disabled temporarily by the user. In addition to the constraints defined in the meta-model the model builder is able to add and remove custom constraints at the modeling level.

The constraint debugger assists the modeler in discovering erroneous constraint definitions. The stack of evaluated expressions along with the current values of all variables are displayed in the debugger. The evaluation tracking facility can be turned off when the designer is confident in the correctness of the constraint definitions.

VISUALIZATION

The modeling framework provides different kinds of graphical interfaces to the model database. The primary display area represents models as separate windows showing contained objects as icons and lines. The physical position of these objects can be arranged arbitrarily and independently in each aspect of the model. The connection paths are controlled by a powerful real-time autorouter. The object positions in different aspects can be selectively synchronized to help clean up large models.

The model browser uses a different visualization approach, displaying the model hierarchy as a tree where non-leaf (composite) nodes can be collapsed or expanded on demand. While this method provides less control and information on a specific node it reveals the whole model hierarchy.

The third interface displays model information in tabular format, similar to a spreadsheet, which supports batch editing and consistency checking nicely.

Finally, it is the main editing window where most of the model creation and modification takes place. Model visualization can be customized to fit the target modeling methodology. All object drawing and mouse handling operations are assigned to a software component, called a decorator, outside of the regular user interface. Whenever these operations need to be executed the GME editor calls the decorator registered for the current modeling language through a predefined interface. GME comes with a default decorator and some samples. Any modeling paradigm can have a custom decorator implementing domain-specific visualization. Decorators have full access to the model database to be able to provide context based visualization for the decorated objects. The graphical framework may send update requests to a specific decorator with a given frequency, if animated visualization is requested.

EXTENSIBILITY

GME was designed with extensibility as one of the most important goals. The tool has a modular component architecture shown in Figure 3. At the bottom, different storage formats,

ranging from relational databases through a fast proprietary binary file format to XML, are supported. Two key components of the GME are GMeta and GModel. The GMeta component defines the modeling paradigm, while GModel implements the GME modeling concepts for the given paradigm. GMeta configures itself by reading the meta specifications (generated from the metamodels), while GModel uses the services of GMeta for self-configuration. The GModel component exposes its services through a set of public interfaces as well. The architecture is based on Microsoft COM technology.

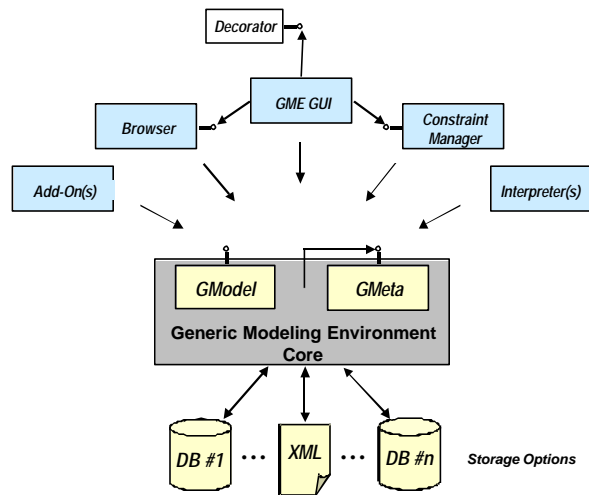


Figure 3 GME Architecture

The user interacts with the components at the top of the architecture: the GME User Interface, the Model Browser, the Constraint Manager, Interpreters and Add-ons. Add-ons are event-driven interpreters. The GModel component exposes a set of events, such as “Object Deleted” or “Attribute Changed,” etc. External components can register to receive some or all of these events. They are automatically invoked by GModel when the events occur. Add-ons are extremely useful for extending the capabilities of the GME User Interface, for example. When a particular domain calls for some special operations, these can be supported by add-ons without modifying any GME components.

Since the event dispatching mechanism is a vital part of the architecture, its performance has a significant impact on the usability of the overall framework. All external components own a so called territory that keeps track of all objects that the component may be interested in. This repository is automatically maintained based on the object references that GModel ever handed over to the component. A specific event will be propagated to the component only if the affected object is in the component’s territory. This technique reduces the number of redundant event messages dramatically.

The performance and reliability of the overall system is further improved with the help of transactions. Model operations – even read-only actions - on the GModel level must be encapsulated in transactions. Components are receiving events when a transaction begins and finishes, thus they are able to aggregate multiple changes and react to those changes at the end of the whole transaction. In case of a read-only transaction they may discard all notifications.

The framework keeps track of all registered external components and integrates them into the user interface. Add-ons are automatically started when a project in a supported paradigm is opened. Interpreters are integrated into the menus and toolbars of the user interface after successful registration. Information on the registered components is retained across invocations of the tool.

COM technology enables the seamless integration of additional components. Moreover, components can be implemented using any programming language that supports COM, such as C++, Visual Basic, Python, Java or C#.

The different standard technologies applied throughout the environment, such as UML, OCL, XML, and COM, ensure maximum flexibility. The modeling tool also provides a C++ programming framework along with a code wizard to help developing external components without understanding COM or working on infrastructural code. The framework, called the Builder Object Framework is a hierarchy of classes that represents and mirrors the model database in the form of C++ objects. It enables the developer to focus on the domain specific part of the program immediately, thus implementing simple but useful components can take as little time as a few hours.

EXAMPLE

As a first example, consider the metamodeling language itself. Figure 4 shows the metamodel of a simple hierarchical finite state machine modeling paradigm (HFSM) in GME configured for metamodeling. In the lower right corner of the GME window you can see the currently active modeling language; in this case it is MetaGME. The window in the lower left corner is the partbrowser. It contains the kind of parts that can be inserted in the current aspect of the currently open model. The tabs in the bottom show all available aspects. For the metamodeling language these are: ClassDiagram, Visualization, Constraints and Attributes. Aspects help manage model complexity by separating orthogonal concerns. Aspects are captured in the Visualization aspect of the metamodel (not shown).

The window in the lower right corner shows the attributes, preferences and properties of the selected object or objects. The window on the top right is the model browser. It shows the hierarchy of the whole project. Notice that in this case it shows parts GuardCondition, Main or UniqueName, that are not shown in the main window. The reason is that those parts are shown in different aspects of the Main model. GuardCondition is an attribute of Transition captured in the Attributes aspect, Main is the single aspect in our HFSM modeling language modeled in the Visualization aspect, while UniqueName is a constraint specified in the Constraint aspect. This constraint specifies that no two substates of a state can have the same name:

self.parts(State)->forall(x, y | x.name = y.name implies x = y)

For this particular constraint the Close Model event seems to be the best candidate for enforcement. Whenever the parent model is closed the constraint will be checked.

The metamodeling environment has its own decorator. It is capable of visualizing the UML class diagram notation. It displays the names, stereotypes and attributes of all classes. It shrinks or expands the box to fit the displayed text. This particular decorator is about 1500 lines of C++ code most of which is the baseline code shared among all decorators.

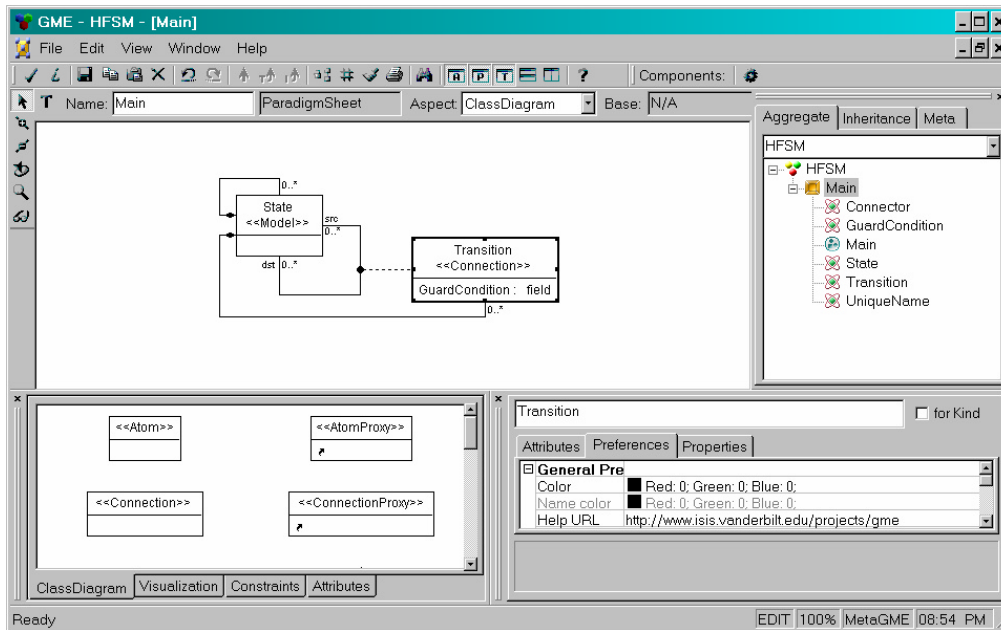


Figure 4 HFSM metamodel

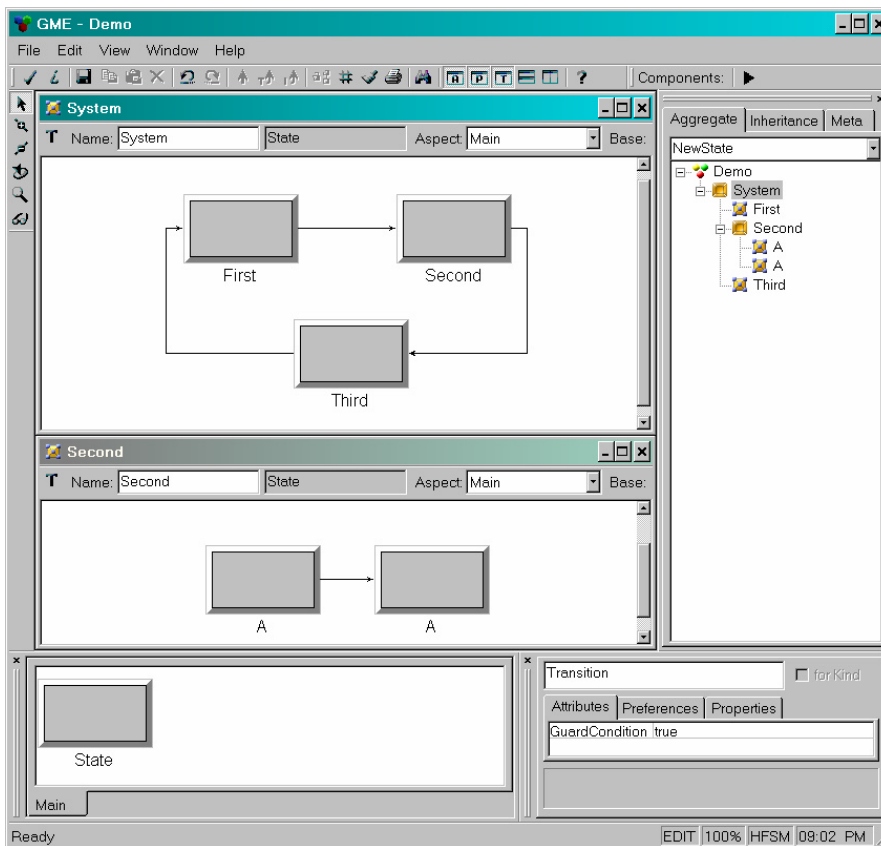


Figure 5 Example HFSM model

The metamodeling environment has a good example for an add-on. OCL syntax checking is a very specific functionality that is not part of the baseline GME program. However, its function is very important to metamodeling; the user does not want to wait to catch syntax errors in her constraints until a constraint is checked in the target environment. Therefore, we provide an add-on to the metamodeling environment that checks the OCL syntax of the constraint that is currently being edited. The OCL expression is a textual attribute of the constraint object. The OCL syntax checker add-on is registered to catch attribute change events. Whenever it is fired it parses the OCL expression and provides immediate feedback to the user.

As most meaningful modeling paradigms, the metamodeling environment has its own interpreter. It parses all the class diagrams (in the HFSM case there is only a single one) and generates an XML representation of the modeling language. GME can read this file and configure itself to support the new language. Figure 5 shows an example model captured in the resulting HFSM modeling environment.

The name of the modeling language is shown in the lower right corner again. Notice that the only part shown in the part browser is State and the only aspect is Main. The attribute window shows the GuardCondition attribute of one of the transitions. Our simple HFSM environment uses the standard, built-in decorator that is able to display boxes, icons, names etc. Notice that the state Second has two substates with the same name “A”. When trying to close the model or explicitly requesting constraint checking, the violation is caught by the constraint manager. The error message displayed is shown in Figure 6.

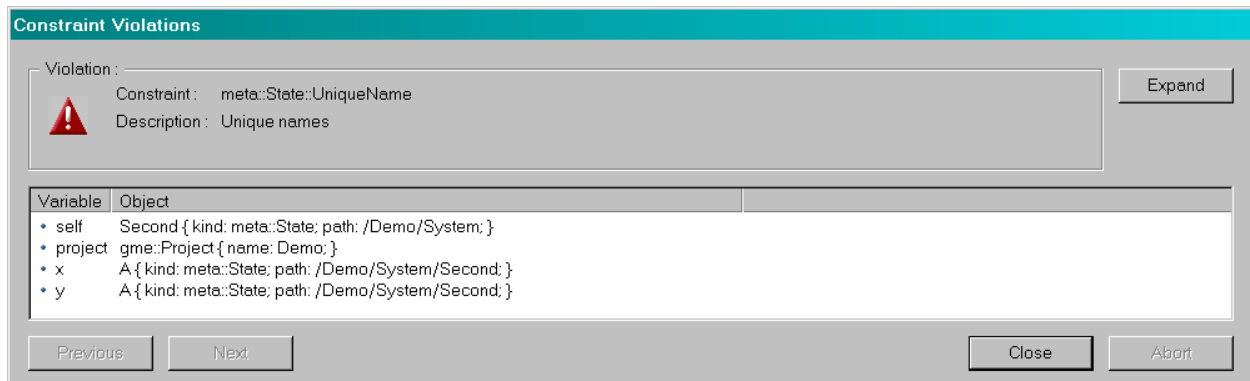


Figure 6 Constraint violation

PRACTICAL APPLICATIONS

Three practical applications of GME are presented below:

MILAN, the Model-based Integrated simuLAtion framework, is a GME-based extensible environment that facilitates rapid evaluation of different performance metrics, such as power, latency and throughput, at multiple levels of granularity, of a large class of embedded systems by seamlessly integrating different widely-used simulators into a unified environment (Ledeczi, Davis, Neema & Agrawal, 2003). 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 provides an integrated environment where existing development and analysis tools, primarily simulators, can work seamlessly together. MILAN defines an integrated data model that captures the shared semantics of all the tools integrated and a bidirectional semantic translator for each of them. This is an elegant solution to tool integration that also avoids the scalability problem associated with pair wise translation, i.e. the integration of a new tool requires only a single new translator (and possibly modifications to existing ones) and not N , i.e. the number of integrated tools. This is a key advantage since one of the design goals of MILAN is to provide an open environment so that users can integrate additional tools on their own.

The complex modeling language allows for the specification of the desired application functionality in the form of an extended dataflow representation with strong datatyping and parametric modeling, provides the means to specify the available hardware resources and enables the user to define mapping information between the two. Finally, application requirements can also be captured by explicit constraints in the models. Instead of specifying a point solution, however, MILAN enables capturing the whole design-space of the application. At any point in the hierarchical dataflow, explicit design or implementation alternatives can be specified. For example, different algorithm choices optimized for speed, memory requirements or power consumption can be captured this way or optimized implementations can be provided for different hardware targets. Similarly, multiple hardware resource options can also be supplied. Finally, hardware/software allocation need not be fully specified. These techniques make it possible to describe a large – potentially exponential – set of solutions forming the design space of the application. Our symbolic design-space exploration and pruning technology rapidly narrows it down to the subset that satisfies all requirements of the system that are captured as constraints (Neema, 2001).

Tools currently integrated into the MILAN framework include such functional simulators as Matlab, SystemC and ActiveHDL (a VHDL simulator) and a high-level performance estimator, HiPerE (Mohatny & Prassana, 2002). These tools are not aware of which parts of the system are to be implemented in hardware and which parts in software. This makes MILAN a true system-level hardware/software codesign environment. Of course, lower-level integrated tools, such as such cycle-accurate simulators as SimpleScalar, are already tied to a specific hardware technology. Note that the different tools only communicate through the integrated system models. This eliminates the need for each tool to be interfaced directly to all others.

SSPF. A predecessor of GME was used to create the Saturn Site Production Flow (SSPF) system that monitors the car manufacturing process at GM's Saturn Corporation providing key production measures to managers in real-time (Long, Misra & Sztipanovits, 1998). The system models describe the manufacturing processes down to the machine level, the buffers between the processes (e.g. conveyor belts), the instrumentation (i.e. PLCs), and how the information is to be presented to the user. The interpreters 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.

GRATIS, a graphical development environment for TinyOS, provides an intuitive visual interface and automatic code generation capability for the development of TinyOS-based sensor network applications (Volgyesi & Ledeczi, 2002). With the original TinyOS tools (Hill, 2000) working with textual configuration files while developing non-trivial applications could quickly become an error-prone and tedious process. Function-like entities have two or more names in the

final application; this characteristic is inherent in the flexible design enabling the creation of countless different applications without touching the implementation of the individual components. However, as a side effect, it has notable impact on the maintainability of the applications. GRATIS replaces the textual representation of the interface and configuration specifications. Even with a simple application, a more expressive representation of components and interconnections between them can help design better applications and increase their readability. With more sophisticated components and especially with hierarchical composition this becomes an absolute requirement. There are cases, where components might impose additional complex restrictions on their use – like mutual exclusion or maximum fan-out – in addition to the normal rules of composition in TinyOS. These additional requirements can be easily captured by the constraint language provided by the GME modeling framework.

Since all practical applications use system components from the TinyOS distribution, GRATIS also provides a mapping from the existing large code base to the graphical environment. Therefore, the interpreter not only generates text files from graphical models, but it is also capable of parsing existing files and building the corresponding GME models from them. The main use of this parsing feature is to automatically generate the graphical equivalent of the TinyOS system components and to provide them as a library to the user in the GRATIS environment. An indisputable benefit of the parsing and model building process is an exhaustive testing, since the parser – with the help of the predefined constraints in the metamodel – builds and validates all components and applications found in the source tree. Since scripting languages are generally superior to compiled languages in the field of text processing, we have implemented GRATIS using GME and the Python language exclusively, which also demonstrates an extension alternative to our C++ interpreter framework.

Other experimental modeling languages have also been implemented to describe not only the type requirements, but temporal dependencies and the implementation details also. These languages comprise our further work to understand compatibility and composability issues better in the field of embedded systems. GME proves to be an efficient tool and approach to build such environments.

COMPARISON TO OTHER TOOLS

In terms of supported features, maturity, and the number of real world applications, three configurable environments stand out: Dome by Honeywell Laboratories (Honeywell, 2000), MetaEdit+ by MetaCASE Consulting of Finland (MetaCase, 2000) and our own Generic Modeling Environment (GME) (Institute for Software Integrated Systems, 2002). The four key areas that enable true support for widely different modeling methodologies are metamodeling, constraint management, visualization and extensibility.

Metamodeling: Meta-modeling may be regarded as just another type of modeling, therefore, Dome and GME use the tools themselves to implement this functionality. MetaEdit+ has a more conservative approach; a series of dialog boxes are used to specify the meta-model in a non-graphical way. Meta-models typically evolve while being used, and modifications in the meta-model often break the validity of models. These concerns are just partially handled in Dome and MetaEdit+. GME is the only tool that demonstrates a strict discipline: meta-models are versioned, and new versions of meta-models do not affect existing models until they are explicitly upgraded to the new version. Such an upgrade implies extensive validity checking. This is somewhat cumbersome, but essential for warranting the correctness of models, especially if they are beyond the usual demo application size.

GME is the only environment that provides support for metamodel composition and metamodel libraries.

Constraint management: Dome and MetaEdit+ have some built-in support for certain types of frequently used constraints. GME, on the other hand, has a full featured constraint manager supporting OCL, a standard constraint language. Constraints can also be associated with editing events and priorities, making constraint management interactive and flexible.

Visualization: While MetaEdit+ provides an impressive built-in symbol editor, GME and Dome provides only the choice of simple built-in symbols and bitmap files provided by the user, and rely on user-defined drawing routines for more complex visualization. While this user-defined visualization can be very powerful and flexible, their implementation obviously requires some traditional programming skills from the user.

Extensibility: the interface to Dome models is primarily through its Alter language, a Scheme variant. MetaEdit+ only supports a proprietary scripting language to access the models. On the other hand, the component based architecture of GME makes it easily extensible. Meta and model information are all available through public COM interfaces. Events are also exposed through COM. When any component makes a change to the models, all other interested components are notified. The toolset can be extended using any programming language that supports COM (e.g. C++, VB, Python). Furthermore, GME supports XML export/import for both model and meta information.

CONCLUSIONS

We presented GME, a framework that enables the rapid development of modeling environments. Its powerful metamodeling capabilities make it possible to create a full-featured modeling environment in hours. For example, an experienced user could easily create the simple HFSM environment presented above as an example in well under an hour. Hence, GME supports the rapid design of modeling languages enabling immediate hands-on experience with the language. It supports an iterative design method of modeling methodologies; users can quickly evolve their language design by iteratively modifying the corresponding metamodel. When the modeling methodology becomes satisfactory then the effort to create decorators, add-ons and interpreters are much better justified. Note, however, that typical domain-specific components for GME are relatively small, hence the effort to create the additional software modules is not prohibitive even for small projects. For example, an HFSM simulator that animates the automaton within the GME user interface was written by a graduate student in a couple of weeks.

The HFSM example presented here is very simplistic for clarity. In our experience, GME scales very well. The modeling language of MILAN, for example, has hundreds of modeling concepts. Both MILAN and GRATIS support large model databases. Other large-scale, real-world applications of the technology are presented in (Ledeczi et al., 2001).

The research in configurable modeling environments, metamodeling methodologies and model visualization continues at our institute. GME is just a reflection of the current state-of-the-art of our research. New versions of the software are regularly released multiple times a year. Ongoing work includes research on generative modeling, automatic interpreter generation using graph transformations (Agrawal & Karsai, 2003) and integrating GME into the popular Eclipse framework (Eclipse.org Consortium, 2001).

ACKNOWLEDGEMENTS

The authors would like to express their gratitude to DARPA and the Boeing Company for the generous sponsorship of the research describe here.

REFERENCES

- Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J. & Karsai, G. (2001). *Composing Domain-Specific Design Environments*. IEEE Computer, 34(11), pp. 44-51.
- Honeywell (2000). *Dome Official Web Site*. Retrieved from <http://www.htc.honeywell.com/dome>
- MetaCase Consulting (2000). *MetaEdit+ Official Web Site*. Retrieved from <http://www.metacase.com>
- Ledeczi, A., Davis, J., Neema, S. & Agrawal, A. (2003). *Modeling Methodology for Integrated Simulation of Embedded Systems*. ACM Transactions on Modeling and Computer Simulation, 13(1), 82-103.
- Ledeczi, A., Nordstrom, G., Karsai, G., Volgyesi, P. & Maroti, M. (2001). *On Metamodel Composition*. IEEE Conference on Control Applications 2001, Mexico City, Mexico, CD-ROM.
- Institute for Software Integrated Systems (2002). *GME Official Web Site*. Retrieved from <http://www.isis.vanderbilt.edu/Projects/gme>
- Eclipse.org Consortium (2001). *Eclipse Official Web Site*. Retrieved from <http://eclipse.org/>
- Agrawal, A. & Karsai, G. (2003). *A UML-based Graph Transformation Approach for Implementing Domain-Specific Model Transformations*. International Journal on Software and Systems Modeling, (Submitted).
- Long, E., Misra, A. & Sztipanovits, J. (1998). *Increasing Productivity at Saturn*. IEEE Computer, 31(8), pp. 35-43
- Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D. & Pister, K. (2000). *System Architecture Directions for Networked Sensors*. Proceedings of Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) 2000, Cambridge, MA, USA.
- Volgyesi, P. & Ledeczi, A. (2002). *Component-Based Development of Networked Embedded Applications*. 28th Euromicro Conference, Component-Based Software Engineering Track, Dortmund, Germany.
- Mohatny, S. & Prassana, V. K. (2002). *HiPerE: A Framework for Rapid System Level Power and Performance Estimation of Embedded Applications on SoC/SoP Architectures*. Design, Automation, and Test in Europe.
- Neema, S. (2001). *System Level Synthesis of Adaptive Computing Systems*. Ph. D. Dissertation, Vanderbilt University, Department of Electrical and Computer Engineering.