Metamodel Composition in the Generic Modeling Environment

Akos Ledeczi, Peter Volgyesi, Gabor Karsai
Institute for Software Integrated Systems, Vanderbilt University
Nashville, TN, 37235, USA
akos@isis.vanderbilt.edu

Introduction

Domain-Specific Design Environments (DSDE) capture specifications and automatically generate or configure the target applications in particular engineering fields. Well known examples include Matlab/Simulink for signal processing [1] and LabView for instrumentation [2], among others. The success of DSDEs in a wide variety of applications in diverse fields is well documented. Their merit lies in the natural interface they provide to the designer. While the output of a DSDE is typically software in some shape or form, the user does not have to be a software engineer. In fact, a DSDE is specifically targeted at the domain engineers who want to keep using the standard notations and diagrams that are natural in their field.

The main advantage of DSDEs is the automatic generation of the target application (software, hardware, configuration files, input to analysis tools, etc.) from specifications in the domain language. This has enormous productivity and system lifecycle cost implications. The alternative, the traditional approach, is to have the specifications provided by the domain engineers translated into code by a software. Clearly, development cost, turnaround time, productivity, maintenance cost and time, and necessary manpower would compare unfavorably to a DSDE. Why are then the use of DSDEs not more widespread?

Unfortunately, the development of a typical DSDE is very expensive. A sophisticated environment can easily take hundreds of man-years to complete. The relatively few commercially available DSDEs are all targeting domains with large markets where the large initial investment is offset by high volume. It is simply not cost efficient to develop DSDEs for narrow domains where only a handful of copies are needed.

To solve this problem, we advocate the idea of a Configurable Domain-Specific Development Environment (CDSDE) that is configurable to a wide range of domains. The CDSDE needs to provide a set of generic concepts that are common to most DSDEs. It is then customized to every new domain to support the given domain language directly. While the development of the CDSDE itself is expensive, it needs to be done only once; the initial investment is spread across multiple domains.

This paper introduces a CDSDE, the Generic Modeling Environment (GME 2000), developed at the Institute for Software Integrated Systems at Vanderbilt University. GME 2000 utilizes metamodeling to define the domain modeling language along with model integrity constraints and it automatically configures itself to support the new language. The metamodeling capabilities of GME 2000, especially its support for metamodel composition, are in the focus of this paper.

The Generic Modeling Environment

The Generic Modeling Environment (GME 2000) is a configurable toolkit for creating domain-specific design environments. The configuration is accomplished through metamodels specifying the modeling paradigm (modeling language) of the application domain. The modeling paradigm contains all the syntactic, semantic, and presentation information regarding the domain.

A reusable framework for creating domain-specific design environments must support a set of abstract modeling concepts that are generic enough to be applicable to a wide range of domains. The choice of these generic concepts is the most critical design decision. GME 2000 supports various concepts for building large-scale, complex models. These include: hierarchy, multiple aspects, sets, references, and explicit constraints. Selected concepts are “instantiated,” i.e. customized, for each target domain, possibly multiple times, to support domain concepts directly. The key idea is the consistent application of a meta-level architecture. GME 2000 follows the standard four-layer metamodeling architecture applied in the specification of CDIF and UML.

GME 2000 architecture

GME 2000 has a modular, component-based architecture depicted in the figure below.

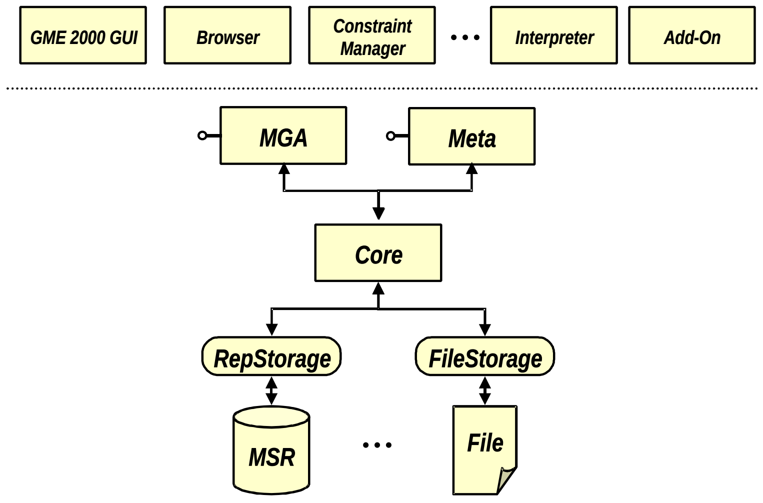


Figure 1. GME 2000 architecture

The thin storage layer includes components for the different storage formats. Currently, MS Repository (an object oriented layer on top of MS SQL Server or MS Access) and a fast proprietary binary file format are supported. Supporting an additional format (e.g. ODBC) requires the implementation of a single, well-defined, small interface component.

The Core component implements the two fundamental building blocks of a modeling environment: objects and relations. Among its services are distributed access (i.e. locking) and undo/redo.

Two components use the services of the Core: the Meta and the MGA. The Meta represents the modeling paradigm, while the MGA implements the modeling concepts for the given paradigm. The MGA uses the Meta component extensively through its public COM interfaces. The MGA component exposes its services through a set of COM interfaces as well.

The user interacts with the components at the top of the architecture: the GME 2000 User Interface, the Model Browser, the Constraint Manager, Interpreters and Add-ons.

Model interpreters are translators that take information captured in the models and generate some output, e.g. source code, configuration files, reports, etc. Add-ons are event-driven model interpreters. The MGA component exposes a set of events, such as “Object Created,” “Set Member Removed,” “Attribute Changed,” etc. External components can register to receive some or all of these events. They are automatically notified by the MGA when the events occur. Add-ons are extremely useful for extending the capabilities of the GME 2000 User Interface. When a particular domain calls for some special operations, these can be supported without modifying the GME 2000 itself.

The Constraint Manager can be considered as an interpreter and an add-on at the same time. It can be invoked explicitly by the user and it is also invoked when event-driven constraints are present in the given paradigm. Depending on the priority of a constraint, the operation that caused a constraint violation can be aborted. For less serious violations, the Constraint Manager only sends a warning message.

The GME 2000 User Interface component has no special privileges in this architecture. Any other component (interpreter, add-on) has the same access rights and uses the same set of COM interfaces to GME 2000. Any operation that can be accomplished through the GUI, can also be done programmatically through the interfaces.

Data access and extensibility

As described previously, GME 2000 is completely component-based with public interfaces among its components to support one of its primary application areas, data and tool integration. Since the component model is COM, the primary languages for integration are C++ and Visual Basic, while Java, Python, etc. access is also available. Access is bi-directional, and fully transactional, which makes different 'on-line modeling' scenarios feasible. For example, the GME 2000 editor itself can be used as the user interface of a generated application to provide feedback to the user in terms of the models. Furthermore, the bi-directional access makes it possible to convert legacy data into models in an automated fashion.

Programming at the component level is somewhat challenging since it requires advanced transaction control and event handling. Several alternatives provide easier access through simpler interfaces (albeit with limited functionality). First, the GME pattern-based report language provides simple reporting capabilities by interpreting macro definitions in a simple text input file. A more complex interface is layered on top of the COM interfaces providing an easy-to-use, extensible C++ API. GME 2000 also provides bi-directional XML access for both model and metamodel information.

Type hierarchy

To support model reuse and maintenance, GME 2000 supports model types, instances and type inheritance. These concepts closely resemble those of object-oriented programming languages. The only significant difference is that in GME 2000, model types are similar in structure and appearance to model instances; they too 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 in a type 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 and subtypes and instances of subtypes all the way down the inheritance hierarchy. There is a set of well-defined rules specifying the exact behavior of type inheritance in GME 2000.

Metamodeling Environment

An interesting aspect of the GME 2000 tool suite is that the same set of tools is used for metamodeling as for domain modeling. The metamodeling problem can be thought of as just another domain, the field of domain-specific design environments. Hence, the metamodeling language is just another domain language. The meta-specifications that configure GME 2000 are generated by the metamodeling translator from the metamodels. The metamodeling environment itself is generated by the same translator when translating the meta-metamodels.

The metamodeling language in GME 2000 is the UML class diagram notation [3]. The metamodels fully specify the domain modeling language, or more precisely, its concrete syntax. They do not, at least not entirely, specify the static semantics of the language. By static semantics we mean the set of rules that specify the well-formedness of domain models. UML class diagrams do allow the specification of some basic rules, for example, the multiplicity of associations. For more complex semantic specifications, however, UML includes the Object Constraint Language (OCL) [4], a textual predicate logic language. GME 2000 adopts OCL as well; metamodels consist of UML class diagrams and OCL constraints.

Metamodel composition

Just as the reusability of domain models from application to application is essential, the reusability of metamodels from domain to domain is also important. Ideally, a library of metamodels of important sub-domains should be made available to the metamodeler, who can extend and compose them together to specify domain languages. These sub-domains might include different variations of signal-flow, finite state machines, data type specifications, fault propagation graphs, petri-nets, etc. The extension and composition mechanisms must not modify the original metamodels, just as subclasses do not modify base classes in OO programming. Then changes in the metamodel libraries, reflecting a better understanding of the given domain, for example, can propagate automatically to the metamodels that utilize them. Furthermore, by precisely specifying the extension and composition rules, models specified in the original domain language can be automatically translated to comply with the new, extended and composed, modeling language.

The GME metamodeling language is based on UML class diagrams. However, to support metamodel composition, some new operators are necessary. The equivalence operator is used to represent the union of two UML class objects. The two classes cease to be separate entities, but form a single class instead. Thus, the union includes all attributes, compositions and associations of each individual class. Equivalence can be thought of as defining the "join points" or "composition points" of two or more source metamodels.

New operators were also introduced to provide finer control over inheritance. When the new class needs to be able to play the role of the base class, but its internals need not be inherited, we use interface inheritance. In this case, all associations and those compositions where the base class plays the role of the contained object are inherited. On the other hand, when only the internals of a class are needed by a subclass, we use implementation inheritance. In this case, all the attributes and those compositions where the base class plays the role of the container are inherited. Notice that the union of these two new inheritance operators is the "regular" UML inheritance as illustrated in the figure below.

Consider the left hand side of Figure 2 with the two UML classes *Base* and *Sub*. Sub is derived from Base through both implementation inheritance (denoted by a filled circle inside a triangle) and interface inheritance (denoted by an empty circle inside a triangle). By applying the interface inheritance operator, we get the equivalent class diagram consisting of *Base2a* and *Sub2a*. Similarly, applying implementation inheritance first, we get *Base2b* and *Sub2b*. Finally, continuing from either one and applying the remaining inheritance operator, we end up with the class diagram of *Base3* and *Sub3*. Notice that this matches exactly the diagram we would get by applying regular UML inheritance to Base and Sub instead of the two new operators.

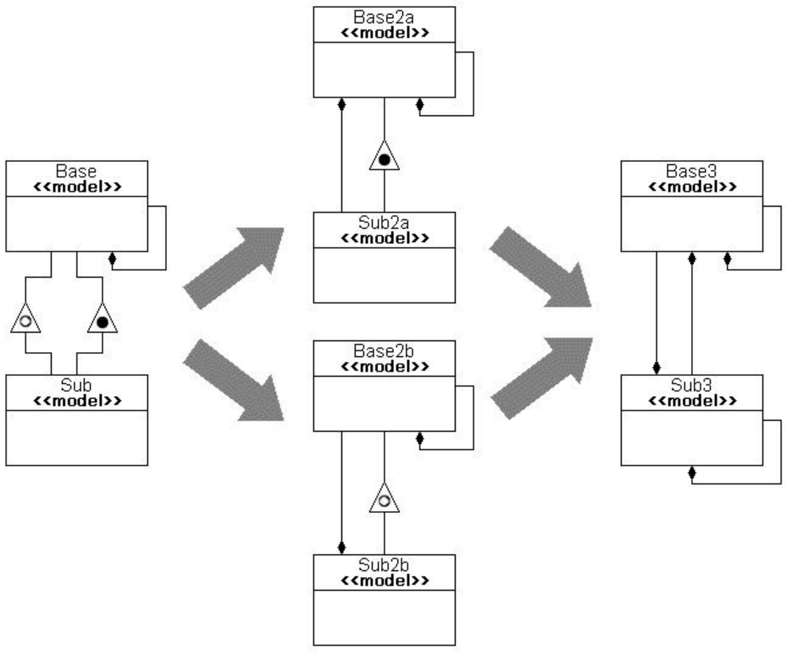


Figure 2 Interface and implementation inheritance

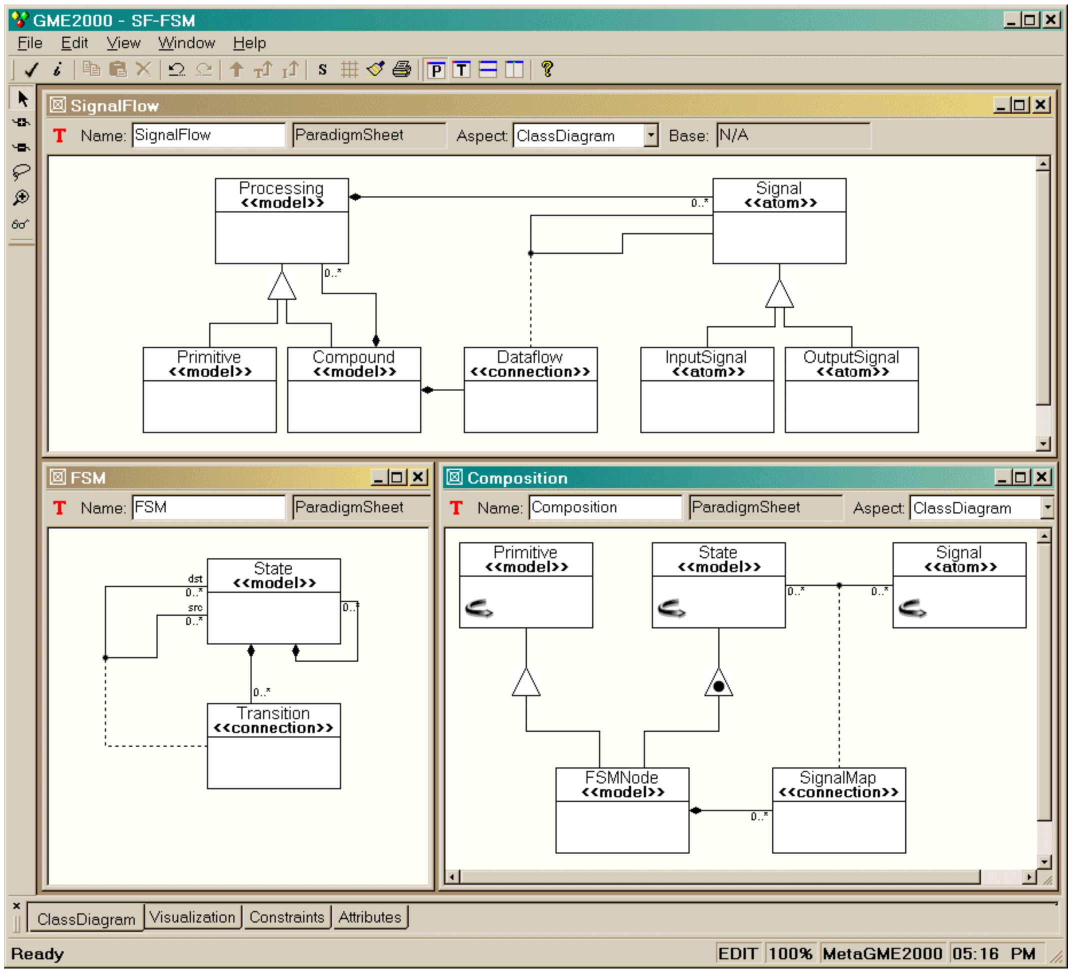
It is important to observe that the use the equivalence and new inheritance operators are just a notational convenience, and in no way change the underlying semantics of UML. In fact, every diagram using the new operators has an equivalent "pure" UML representation, and as such, each composed metamodel could be represented without the new operators. However, such metamodels would either need to modify the original metamodels or require the manual repetition of information in them due to the lack of fine control over inheritance. These metamodels would also be significantly more cluttered, making the diagrams more difficult to read and understand.

Illustrative example

It is probably best to illustrate these ideas through an example. Figure 3 shows three metamodels. The first (*SignalFlow*) specifies a hierarchical signal flow modeling language. *Processing* is an abstract base class. *Compound* is a composite model that can contain other Compounds and *Primitives*. Primitives are the leaf nodes that implement the elementary computation in the graph. (They may have an implementation associated with them in a traditional programming language, for example.) The signal flow connections are implemented by connecting *InputSignals* and *OutputSignals* together with *Dataflow* connections.

The second metamodel (*FSM*) describes a simple hierarchical finite state machine paradigm. *States* can contain other States that can be connected together by *Transition* connections.

We assume that these are existing metamodels that we imported from a metamodel library. We would like to combine them together according to the following rules. We would like to have a new kind of Primitive (*FSMNode*) that can contain a finite state machine specifying its implementation. However, we do not want a State to be able to contain this new kind of model. Furthermore, we want to make selected InputSignals and OutputSignals of any FSMNode to be mapped to certain States it contains using connections. (This could mean, for example, that the data values associated with those signals are accessible from the implementation associated with the given State.)


GME2000 - SF-FSM

SignalFlow — Name: SignalFlow | ParadigmSheet | Aspect: ClassDiagram | Base: N/A

FSM — Name: FSM | ParadigmSheet

Composition — Name: Composition | ParadigmSheet | Aspect: ClassDiagram

Ready | EDIT | 100% | MetaGME2000 | 05:16 PM

Figure 3 Composed Signal Flow and FSM Metamodels

These rules are accomplished by the third metamodel (*Composition*). The new FSMNode class inherits from both Primitive and State. Notice that the curved arrow inside these classes indicate that they are proxies to existing UML classes defined elsewhere. Inheriting from State through the standard UML inheritance would mean that a State could contain an FSMNode violating one of our rules. Instead, we use implementation inheritance that accomplishes exactly what we want: an FSMNode can contain whatever a State can, but it cannot act as a State; it cannot be inserted into a State. (Neither can FSMNodes connected together by Transitions.) Notice how the new SignalMap connection connecting States and Signals is also introduced in the Composition metamodel.

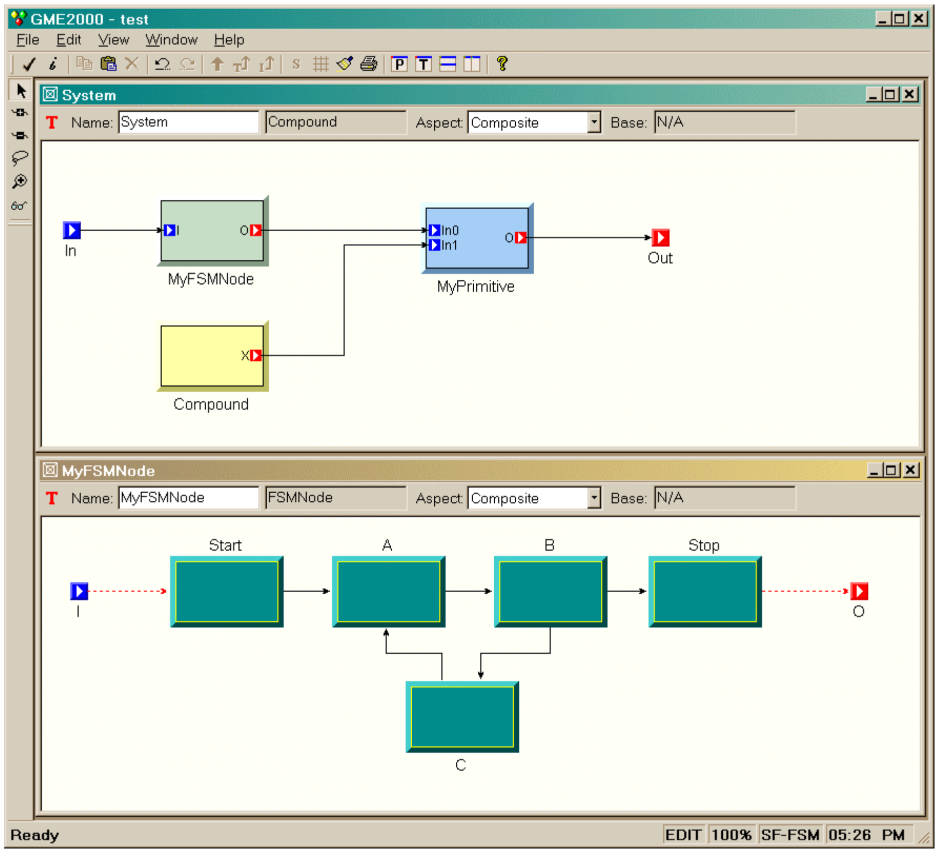


Figure 4 Combined Signal Flow and FSM Models

Figure 4 shows a simple model in the target environment. The System model contains a Compound, a Primitive and an FSMNode. The bottom windows shows the contents of the latter: a simple state machine with Signals mapped to certain States.

Conclusions

We introduced the Generic Modeling Environment (GME 2000), a configurable domain-specific design environment. Other similar environments include Dome by Honeywell Research [7] and MetaEdit+ by MetaCASE Consulting. [6] presents a brief comparison of these three environments.

One of the unique features of GME 2000 is its UML class diagram based metamodeling environment. Our experience showed that some extensions to this standard notation were necessary, primarily to support metamodel composition. The composable metamodeling environment is a brand new addition to GME 2000. We are in the process of applying it to several real world application domains. An early indication of its usability is the significantly increased readability of its own meta-metamodels.

References

[1] http://www.mathworks.com/products/dsp_comm/syslevel.shtml

[2] http://www.ni.com/labview/what.htm

[3] UML Summary, v. 1.0.1, Rational Software Corporation, March, 1997

[4] Object Constraint Language Specification, v. 1.1, Rational Software Corporation, et al., Sept. 1997.

[5] GME 2000 Users Manual. http://www.isis.vanderbilt.edu/projects/gme/Doc.html

[6] A. Ledeczi et al.: "The Generic Modeling Environment," Proceedings of WISP 2001, May, 2001

[7] Dome Official Web Site, Honeywell, 2000, http://www.src.honeywell.com/dome/

[8] MetaEdit+ Official Website, MetaCase Consulting, http://www.metacase.com