Composition and Cloning in Modeling and Meta-Modeling

Gabor Karsai, Miklos Maroti, Akos Ledeczi, Jeff Gray, *Member, IEEE*, and Janos Sztipanovits, *Fellow, IEEE*

***Abstract*—The Generic Modeling Environment (GME) is a configurable tool suite that facilitates the rapid creation of domain-specific model-integrated program synthesis environments. There are three characteristics of the GME that make it a valuable tool for the construction of domain-specific modeling environments. First, the GME provides generic *modeling primitives* that assist an environment designer in the specification of new graphical modeling environments. Second, these generic primitives are specialized to create the domain-specific modeling concepts through meta-modeling. The meta-models explicitly support composition enabling the creation of composite modeling languages supporting multiple paradigms. Third, several ideas from prototype-based programming languages have been integrated with the inherent model containment hierarchy, which gives the domain expert the ability to clone graphical models. This paper explores the details of these three ideas and their implications.**

***Index Terms*—Computer-aided software engineering, software modeling, software prototyping, visual languages.**

I. Introduction

MODEL-INTEGRATED computing (MIC) has been developed over a decade at Vanderbilt University, Nashville, TN, for building embedded software systems. It is an approach to developing systems that directly addresses the problems of system integration and evolution by providing rich domain-specific modeling environments. This technology is used to create and evolve integrated multiple-view models using concepts, relations, and model composition principles routinely used in the domain-specific field [14]. MIC also facilitates systems/software engineering analysis of the models and provides for the automatic synthesis of applications from the models. The approach has been successfully applied in several different applications, including automotive manufacturing [17], signal processing [21], and electrical utilities [18], to name a few.

A core tool in MIC is the Generic Modeling Environment (GME), a derivative of earlier research on domain-specific visual programming environments [9]. GME is a domain-specific modeling environment that can be configured and adapted from meta-level specifications [19]. Thus, based upon the meta-models specifying a domain-specific modeling language, GME can be adapted quickly from a general-purpose drawing tool to a *domain-specific tool* that provides an environment, for example, for modeling an automotive manufacturing plant [17].

There are three characteristics of the GME that make it a valuable tool for the construction of domain-specific modeling environments.

1) The GME provides generic modeling primitives that assist an environment designer in the specification of new graphical modeling environments.
2) These generic primitives are specialized to create the domain-specific modeling concepts through meta-modeling. The meta-models explicitly support composition enabling the creation of composite modeling languages supporting multiple paradigms.
3) Several ideas from prototype-based programming languages have been integrated with the inherent model containment hierarchy, which gives the domain expert the ability to clone graphical models.

Let us informally introduce these concepts through a simple example. Fig. 1 shows the meta-model of a simple modeling language composed of a hierarchical finite-state machine (FSM) representation and a signal flow language.

The meta-modeling approach supported relies on the use of Unified Modeling Language (UML) class diagrams [2]. Meta-models should be able to specify the syntax and semantics of a modeling language. Our meta-models specify the abstract syntax of models in the form of a UML class diagram: the models in the modeling language are instances of classes introduced on the UML class diagram, i.e., the meta-model. In a sense, the UML class diagram specifies a "grammar" that represents all the possible models created in the "language." The UML class diagram also expresses restrictions on models (e.g., cardinality of objects appearing in relations), and we also allow the Object Constraint Language (OCL)-based constraints for specifying well-formedness conditions for the models. These restrictions and constraints specify the static semantics of the modeling language. In the example, the top window shows the meta-model specifying a hierarchical signal flow modeling language. It uses the basic modeling concepts, such as atoms, models, and connections. The modeling concepts are expressed as stereotypes of specific classes and are directly supported by the modeling environment. *Atoms* are classes of objects that do not contain other objects, while *models* are container classes, and *connections* are associations that are

Manuscript received May 21, 2002. Manuscript received in final form April 22, 2003. Recommended by Associate Editor P. Mosterman. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA)/Information Exploitation Office (IXO) Model-Based Integration of Embedded Software (MoBIES) and Software Enabled Control programs.

G. Karsai, M. Maroti, A. Ledeczi, and J. Sztipanovits are with the Institute for Software Integrated Systems (ISIS), Vanderbilt University, Nashville, TN 37235 USA (e-mail: gabor.karsai@vanderbilt.edu; miklos.maroti@vanderbilt.edu; akos.ledeczi@vanderbilt.edu; janos.sztipanovits@vanderbilt.edu).

J. Gray is with the Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL 35294 USA (e-mail: gray@cis.uab.edu).

Digital Object Identifier 10.1109/TCST.2004.824311

1063-6536/04$20.00 © 2004 IEEE

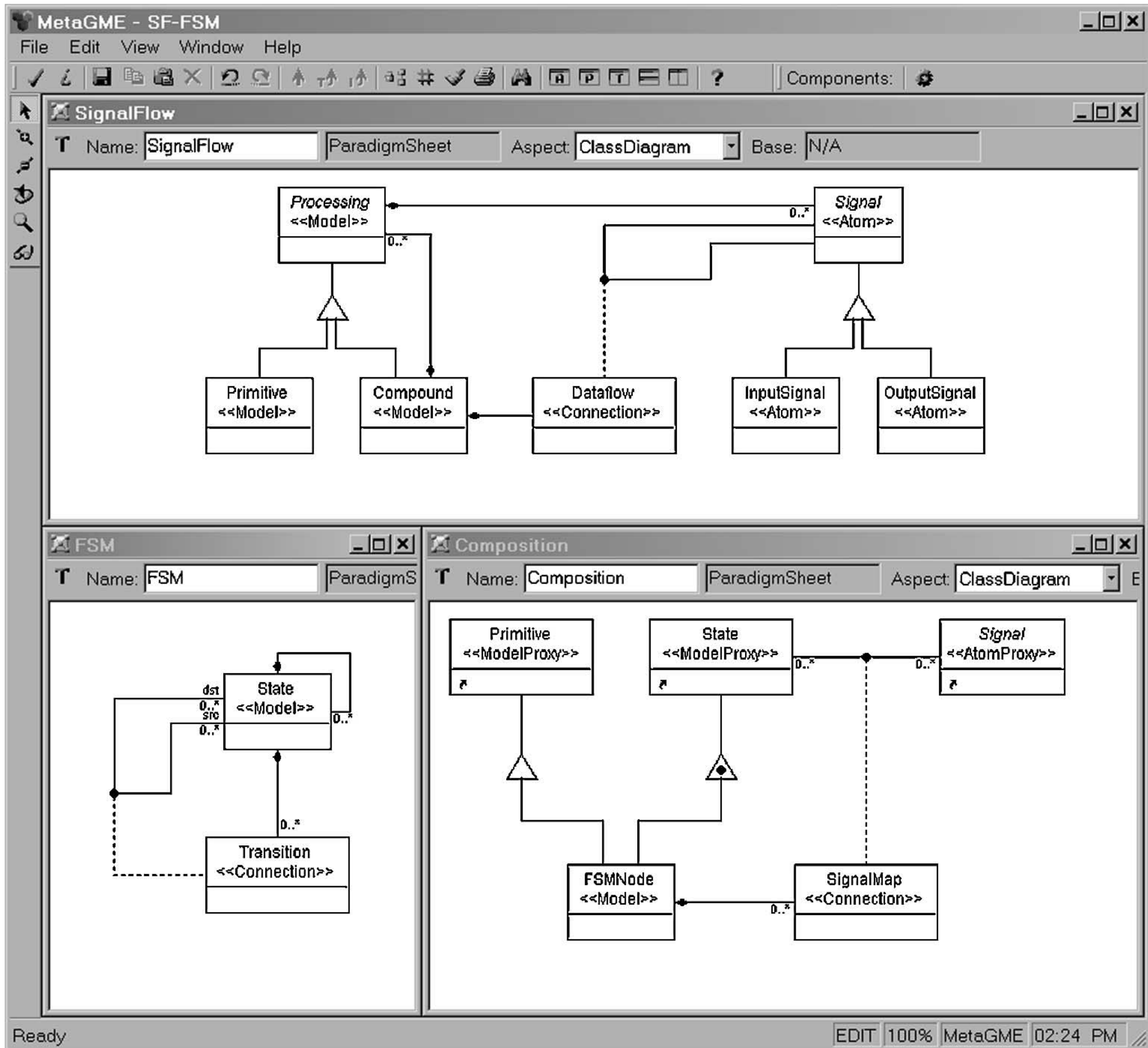

Fig. 1. Meta-model composition.

visualized with a line between iconic objects (which are atoms or models). *Processing* is an abstract base class. A *compound* is a composite model that can contain other compounds and primitives. *Primitives* are the leaf nodes that implement the elementary computation in the signal flow graph. (They may have an implementation associated with them in a traditional programming language, for example.) The signal flow (SF) connections are implemented by connecting InputSignals and OutputSignals together with Dataflow connections.

The second meta-model (FSM) describes a simple hierarchical finite-state machine language. States can contain other states that can be connected together by Transition connections. It can be assumed that both of the SF and FSM meta-models were pre-existing in a meta-model library.

We would like to then compose them according to the following rules. We would like to define 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.)

This composition is defined by the third meta-model (Composition). The new FSMNode class inherits from both Primitive and State. (Notice that the curved arrows inside these classes indicate that they are references 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—an extension to the traditional UML concept of in-

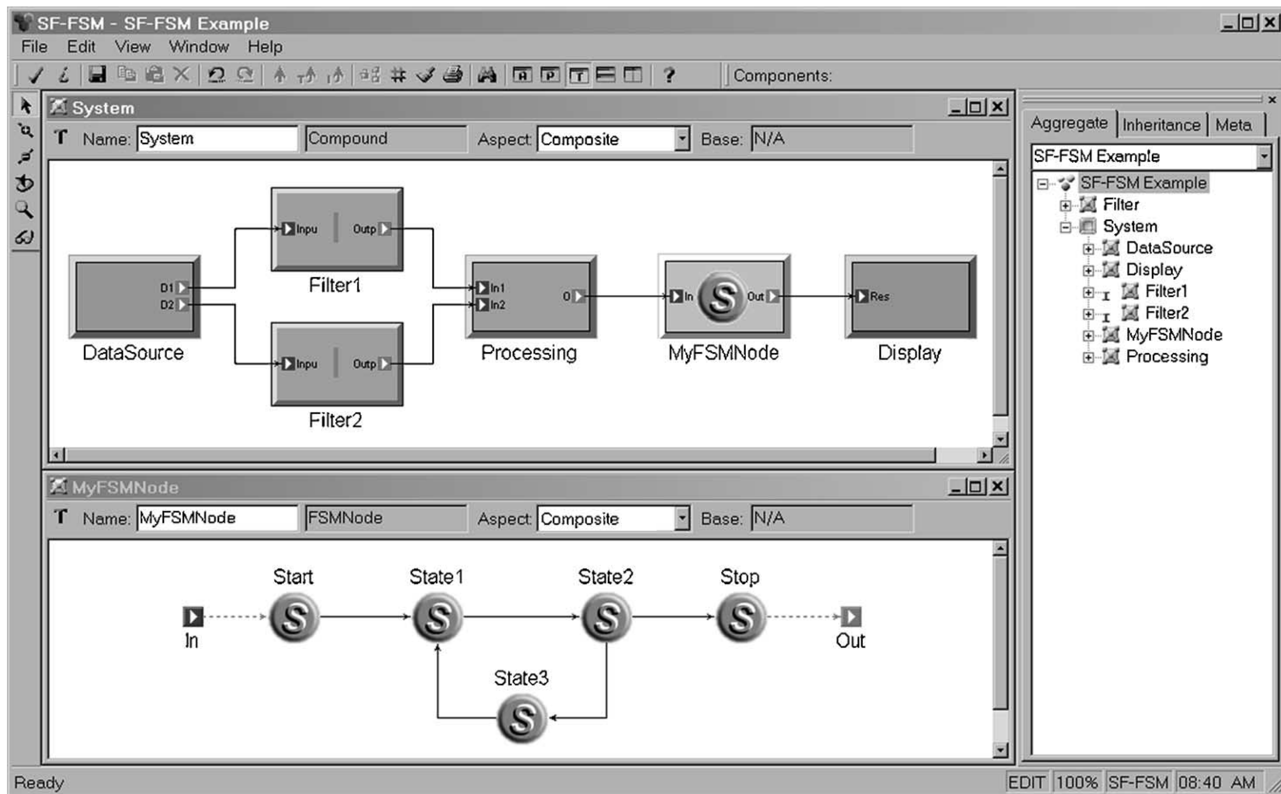

Fig. 2. Sample model in the SF–FSM language.

heritance—(explained in Section V of the paper) which 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. (In addition, FSMNodes also cannot be connected together by Transitions.)

Notice how the new SignalMap connection connecting States and Signals is also introduced in the Composition meta-model. Since FSMNode inherits from Primitive, it can contain Signals. FSMNode also contains a SignalMap that enables connecting Signals to States satisfying our last requirement.

Notice that the meta-models specify the syntax and the static semantics of the modeling language. They do not specify the dynamic semantics. The names FSM, State, and Transition, for example, imply the intentions of the meta-modeler, but how such a model should be executed is not defined here. Assigning dynamic semantics to domain-specific models is the job of separate software components, called model interpreters, in our framework.

Fig. 2 shows an sample model in the modeling environment defined by the above meta-model. The System model contains a signal flow diagram containing a finite-state machine (MyFSMNode). Its structure is shown in the bottom window. A capability called *cloning* is also used in this example. Cloning means that it is sufficient to define a "master" copy—called the prototype—of a model and reuse it in different contexts by creating clones of it[1]. Filter is a prototype Primitive model (shown in the model browser on the right-hand side (RHS) of the figure contained at the root level). System contains two clones of it: Filter1 and Filter2. Any changes to Filter would propagate automatically to both clones. Using simple copies instead of cloning would require manual changes to all three Filter models.

[1]The prototype/clone relationship is reminiscent of the class/instance relationship used in object-oriented systems, but it has somewhat different semantics, to be discussed subsequently in the paper.

This simple example illustrates the key concepts behind GME. A complex modeling task often requires the leveraging of knowledge and expertise in numerous scientific and engineering disciplines. The successful use of an environment like the GME necessitates the collaboration and the skillful execution of the roles of domain expert, environment developer, and experienced programmer. The participants in these roles must synergistically work together in several ways, as explained in the following paragraphs.

Domain Expert: The role of the domain experts is to construct the domain-specific models. They need not possess intricate knowledge of the GME. They only need a basic familiarity that would allow them to create and navigate around the models. They do, however, require detailed insight into the various minutiae of the underlying domain.

Environment Designer: The creation of the domain-specific meta-model, which represents the description of a particular modeling environment, is an arduous task. The meta-model must contain all of the concepts that the domain expert needs to create a model. The individuals responsible for this role must have an understanding of the specific domain, as well as an appreciation of the GME, especially its meta-modeling environment. While the modeling environment creation is fully automated, the development of the domain-specific program synthesis code is still a manual procedure. This is also the responsibility of (some of the) environment designers. This participant must wear two different hats—part programmer and part domain expert.

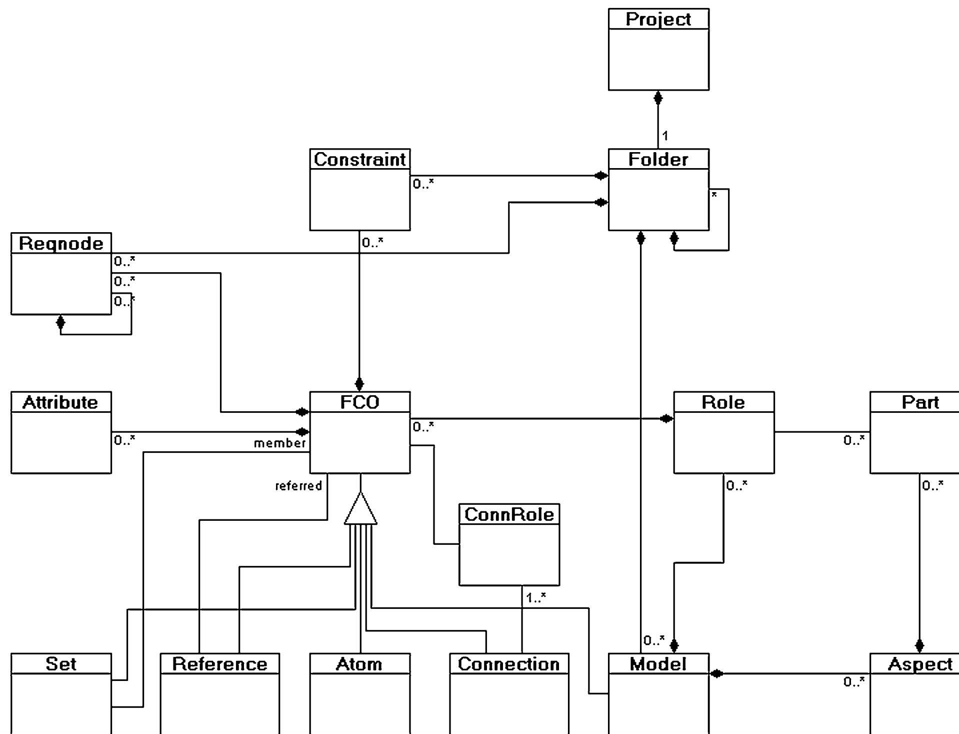

Fig. 3. GME modeling concepts.

GME Developers: The GME developers are unique in that they need not possess any knowledge of a specific domain in which the GME will be applied. They must, however, have a great understanding of general modeling concepts and how those concepts are implemented in a programming language.

This paper describes certain concepts applied in GME that help reduce the complexity involved in the roles of environment designer and domain expert. In the next section, an overview is given of the general modeling concepts available in the GME. The core focus of this paper is the description of meta-modeling, cloning, and compositional meta-modeling. These concepts will be explained in Sections III, IV, and V, respectively, followed by conclusions.

II. General Modeling Concepts

The purpose of this section is to describe the core modeling concepts provided within the GME and their relationships. The GME supports various techniques for building large-scale complex models. The techniques include the following:

- containment hierarchy;
- multiple-perspective views;
- module interconnections;
- sets;
- references;
- explicit constraints.

The UML diagram below depicts a meta-model for these concepts (essentially, the internal data model of GME) and the complex relationship among them. The modeling techniques are implemented with the help of these data structures.

From Fig. 3, it can be seen that the root container class is called *Project*, and a *Project* contains a single *Folder*. Folders are containers that help organize models, just like folders in a file system help manage files. Folders contain *Models*, which are the most fundamental composite modeling elements. First Class Objects (FCOs) are as follows:

- models;
- atoms;
- references;
- connections;
- sets.

Models are the compound objects in our framework. Each FCO contained in a Model always has a *Role* that indicates what purpose the embedded FCO has in that model. The modeling language determines what *type* of objects are allowed in which models in what roles, but the modeler determines the *specific* instances and number of FCOs a given model contains.

The only difference between Models and Atoms is that the latter are elementary objects; they contain no parts. All FCOs have a predefined set of *Attributes*, which are values of a simple type, such as integer and string. The Attribute values are user changeable.

One of the novel modeling concepts in GME is the introduction of *Aspects* [22]. Aspects help manage the complexity of large models by allowing domain experts to focus on selected parts of a design. Aspects are used for visibility control, but they have a specific semantics. Every Model has a predefined set of Aspects, and each part of that Model can be visible or hidden in an Aspect. Every part has a set of primary aspects where it can be created or deleted. There are no restrictions on the set of Aspects a Model and its parts can have; a mapping can be defined to specify which Aspects of a part are shown in a particular Aspect of the parent Model.

It should be noted that "GME aspects" are different from the notion of "aspects" within Aspect-Oriented Programming

(AOP) [12]. Aspects, in the GME sense, can be compared to the viewpoint concept that has been a frequently researched topic within requirements engineering [7], [20]. The notion of views/viewpoints is a key part of the *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems* [8]. This powerful construct, which assists the modeler in separating the concerns of multi-perspectives, is also similar to views in a database [4]. In separate research, we have also investigated the application of aspects, as defined within the context of AOP, to domain-specific modeling [6]. The term, however, as used in the remainder of this paper is aligned more with the notion of a viewpoint.

The modeling language can specify that instances of certain types of FCOs appear on the outside interface of the container model as ports. The primary purpose of ports is to enable Connections to Models. The simplest way to express a relationship between two objects in the GME is with a Connection. Connections can be directed or undirected. In order to make a Connection between two objects, each object must have the same parent in the containment hierarchy. The participating objects also must be visible in the same Aspect (i.e., one of the primary Aspects of the Connection). The meta-model can define several different types of Connections. It also specifies what type of object can participate in a given type of Connection. A signal flow language provides a good example. Signal flow Models contain input and output signal Atoms, and they appear as input and output ports on their outside interface. Signal flow Connections can only be created between input and output signals and input and output ports. Connections can further be restricted by explicit *Constraints* specifying their multiplicity, for instance. Constraints are Boolean expressions, which are evaluated over the object instances. The GME permits the specification of constraints using a variant of the OCL [24].

A Connection can only express a relationship between objects contained by the same Model. In the authors' experience, it is often necessary to associate different types of model objects in different parts of the model hierarchy, or even in different model hierarchies altogether. *References* support these types of relationships well.

References are similar to pointers in object-oriented programming (OOP) languages. A *reference* is not a "real" object; it just refers to one. In GME, a reference must appear as a part in a Model. This establishes a relationship between the Model that contains the reference and the referred object. Any FCO, except for Connections, can be referred to. A Reference can even be referred to by another Reference. References can be connected just like regular model objects. Model references obtain copies of the ports of the referred Model. These ports can then participate in Connections. A Reference always refers to exactly one object, but a single object can be referred to by multiple References.

Connections and References model relationships between (at most) two objects. *Sets* can be used to specify a relationship among a group of objects. The only restriction is that all the members of a Set must have the same parent and be visible in the same Aspect.

A modeling language is defined in terms of Folders, FCOs, (i.e., Models, Atoms, Sets, References, and Connections), Roles, Constraints, and Aspects that are used to build its models. In other words, *the modeling language is represented by the instances of these concepts*. An analogy can be made between this and class-based OOP languages, such as Java, where the corresponding concepts are the class, interface, built-in types, etc. An example is the use of the Model class in a meta-model, which means that the corresponding modeling language supports models of the type denoted by the instance of the Model class. These model definitions act like class definitions in Java: objects that are patterned after them can be created. Furthermore, when a particular domain-specific model is created in GME, it becomes a prototype: a class on a lower abstraction level. This prototype defines a set of models, which are similar to each other in terms of their structure and their attributes, but each one is an individual. Thus, the prototype can be reused, through cloning, as many times as the modeler wishes.

To demonstrate the power of types and prototypes in modeling, reflect on the following example. Consider a model for a large network of components that represents the circuit topology for an electrical utility company. In such a model, it is reasonable to assume that a large number of common components would exist (e.g., transformers). Without prototyping, the potential evolution and manageability of the model might suffer. To illustrate this, consider a case where a slight change to the properties of all transformers is needed. With prototyping, the change could be isolated to one place—the prototype description. If prototyping is not available, however, a modeler must go to every single instance of each transformer and make the same change to each one.

III. The Meta-Model: Types

The basic modeling techniques of GME are generic building blocks that manifest a set of modeling patterns and practices for a large number of domains. They were designed as an aid for graphical and hierarchical modeling of complex engineering systems and to support automatic system synthesis [1], [14], [22]. The domain-specific building blocks are specialized basic modeling elements, together with clear semantics. For example, imagine a simple signal-processing domain that captures the basic concepts related to signal flow. Note that this modeling language is an enhanced version of the SF language introduced in Fig. 1. In this domain, suppose that there are two types of models (Compounds and Primitives, the latter representing leaf nodes in the graph) and two types of atoms (Signals and Parameters). Clearly, these elements have different semantics and constraints; for instance, a Primitive model shall not contain other models. We call this configuration of GME the *meta-model* of the domain. The meta-model contains the definition of the various types of domain-specific modeling elements, together with the specification of their relationships [10].

In this section, the structure of a meta-model is described in more detail. The Extensible Markup Language (XML) has been chosen for the textual representation of meta-models [26]. The XML will be used frequently to illustrate concepts throughout this section. Note, however, that the user of GME does not have to specify the meta-models using the XML representation. There is a GME configuration for a meta-modeling domain, where UML class diagrams are used to specify the meta-models (see Fig. 1). In other words, GME supports a modeling language based on

```
<paradigm name="SignalFlow">
    <meta-model name="Compound"/>
    <meta-model name="Primitive"/>
    <meta-atom name="Signal"/>
    <meta-atom name="Parameter"/>
    <meta-connection name="DataflowConn"/>
</paradigm>
```

Fig. 4. SignalFlow paradigm description.

```
<meta-model name="Compound">
  <role name="Compounds"
        type="Compound"/>
  <role name="Primitives"
        type="Primitive"/>
  <role name="InputSignals"
        type="Signal"/>
  <role name="OutputSignals"
        type="Signal"/>
  <role name="Parameters"
        type="Parameter"/>
  <role name="DataflowConns"
        type="DataflowConn"/>
</meta-model>
```

Fig. 5. Model containment description.

UML class diagrams, which is used to specify meta-models. A model interpreter then automatically generates the XML representation of the target modeling language from these meta-models. This meta-modeling environment is specified within itself through a set of meta-meta-models, allowing the GME to "self-boot." The meta-modeling environment is beyond the scope of this paper (see [13], [14], and [19] for more details).

A. Containment Hierarchies, Types, and Roles

The GME has two distinct groups of models forming two (containment) hierarchies, namely, the meta-model (developed by the environment designer) and the application model (built by the domain expert). Both containment hierarchies form a tree with root objects called the *paradigm* (the *meta-project*) and the *project*, respectively. Each modeling element of the application model has a corresponding meta-modeling element, called its *type*. For example, an "AnalogSignal" modeling element would be associated with a meta-atom named "Signal" that represents its type.

The first role of the meta-model is to describe the different types of modeling elements that domain experts can use. The description of the modeling elements for the signal flow example is shown in Fig. 4.

Merely listing the available types of elements does not specify how they can be composed into a tree structure. Most modeling elements simply cannot contain others. For instance, atoms and connections are not containers. For models, we must list the other types of elements that they may contain. This is best illustrated with the next example meta-model in Fig. 5.

Notice that a single type of atom (Signal) can play different roles (InputSignal or OutputSignal) in instances of the Compound meta-model. This extra indirection makes it easy to reuse the same type more than once by using different role names. The role name is analogous to the name of class member variables in OOP. In our case, however, it does not indicate the presence of a single instance of a type, but a possibly empty collection of instances of the same type.

```
<meta-connection name="DataflowConn">
  <conn-joint>
    <pointer-spec name="src">
        <pointer-item desc="InputSignals"/>
    </pointer-spec>
    <pointer-spec name="dst">
        <pointer-item
            desc="Compounds InputSignals"/>
        <pointer-item
            desc="Primitives InputSignals"/>
    </pointer-spec>
  </conn-joint>
  <conn-joint>
    <pointer-spec name="src">
        <pointer-item
             desc="Compounds OutputSignals"/>
        <pointer-item
             desc="Primitives OutputSignals"/>
    </pointer-spec>
    <pointer-spec name="dst">
        <pointer-item
             desc="OutputSignals"/>
        <pointer-item
             desc="Compounds InputSignals"/>
        <pointer-item
             desc="Primitives InputSignals"/>
    </pointer-spec>
  </conn-joint>
</meta-connection>
```

Fig. 6. Connection and Pointer Specification of DataFlow.

B. Connections and Pointer Specifications

As mentioned in Section II, FCOs can be joined by connections, but not by arbitrarily cutting through the containment hierarchy. The graphical model editor imposes this practical limitation since it has to display each connection in a single model, called the *parent* of the connection. The FCOs can be immediate children or grandchildren of the model. In the latter case, the FCO is displayed inside one of the children of the model as a *port*. This setup makes the interface (i.e., the ports) of composite models visible and manageable. An instance of a meta-connection connects two FCOs, which can be of several types. More precisely, the meta-connection can control the type and the role of the source and destination. In the case of the port, it can also specify the type and the role of the parent of the port. We use pointer items, pointer specifications, and connection joints to specify fully all possible combinations in the following way.

A pointer item is a string that identifies objects of a specific type and role. In the case of an immediate child, this string is simply the role name of an atom, like "InputSignals." In the case of a port, it is a pair of words, where the first word is the role name of the model, and the second is the role name of the atom. For example, the string "Primitives InputSignals" in Fig. 6 matches all InputSignal ports of Primitive children in the parent model in which we are creating the connection. Notice that each word of the pointer item must be a role name and that they uniquely identify the corresponding types of objects, once the parent model of the connection is known.

A pointer specification (pointer-spec) is a list of pointer items along with a name (for a connection, it is either "src" or "dst"). It identifies several types of objects that can act as the "source" or "destination" of a connection. A connection joint ("conn-joint") is just a pair of pointer descriptions that specify all combinations

from “src” objects to “dst” objects. Finally, a meta-connection contains a list of connection joints. We illustrate these concepts by the next example shown in Fig. 6, which captures the following rules:

- InputSignals can be forwarded into contained Compound and Primitive models by connecting them to InputSignal ports.
- OutputSignal ports can be connected to OutputSignal atoms, making them available higher up in the hierarchy.
- OutputSignal ports can be connected to InputSignal ports, forming the interconnections between the contained elements.

Although these concepts might seem cumbersome, they actually help specify all pairs of valid sources and destinations in a precise, compact, and intelligible way. In the previous example, it is important that InputSignal atoms shall not be connected to OutputSignal atoms, a fact that can be very easily missed by humans in a simple enumeration of all valid pairs.

C. References and Sets

Because connections cannot cut through the containment hierarchy, another modeling concept is needed that can link to objects deep in the hierarchy, or in other distant parts of the hierarchy. Recall from Section II that this is achieved with references in the GME terminology. A reference can either point to a single object at a time or be empty. From every other respect, it is very similar to an atom. We already have the required machinery, the pointer specification, to control the possible types of the referred object. Here the first word of each pointer item starts with a type name instead of a role name. This is because references link deep into the hierarchy, where a single role name does not make sense. In the case of connections, the specification always started implicitly from the parent model of the connection, so the type was implied.

The last modeling element that specifies a relation is the Set. It is unique because it can link to several elements at the same time, but they have to be its siblings. In effect, a Sset selects a subset of its siblings. Similar to the meta-reference, the meta-set has a single pointer specification that controls the possible types and roles of siblings to which the set can link.

D. Attributes and Value Types

An attribute of a FCO contains a single value of a predefined value type. The meta-attribute defines the name, the value type, and default value of the attribute. Aside from basic predefined data types, enumerated types are also available. The enumeration type is analogous to the facility offered in most programming languages, where the possible choices of values are provided.

E. Aspects and Parts

Recall that the children of a model are separated according to their roles. The GME visualizes models through aspects by displaying only those children objects that belong to selected groups of roles. This (possibly overlapping) partition of roles into aspects allows multi-perspective views to be projected onto the model.

```
<meta-model name="Compound">
  ...
  <aspect name="SignalflowAspect">
    <part role="Compounds" primary="yes"/>
    <part role="Primitives" primary="yes"/>
    <part role="InputSignals" primary="yes"
          port="yes"/>
    <part role="OutputSignals" primary="yes"
          port="yes"/>
    <part role="DataflowConns" primary="yes"/>
  </aspect>
  <aspect name="ParameterAspect">
    <part role="Compounds" primary="no"/>
    <part role="Primitives" primary="no"/>
    <part role="Parameters" primary="yes"
          port="yes"/>
  </aspect>
</meta-model>
```

Fig. 7. Specification of Aspects.

Each meta-model has a list of aspects, and each aspect contains a list of parts. A part selects a single role of the meta-model (and therefore the corresponding type), together with some auxiliary information. We say that a part is *primary* if the children of the corresponding role are able to be modified in the graphical model editor in the given aspect. Otherwise, we call it *secondary*, which means that they are visible but cannot be modified in that aspect. Any part can be a *port*. This is the place where we choose and control the interface (that is, the ports) of models. These concepts are illustrated in Fig. 7.

F. Registry Nodes, Extensibility, and Names

Environment developers have to configure the various tools of the GME in complete detail to create a coherent environment for domain experts. This includes, for example, the specification of visualization information, such as the color, style, and icon for each meta-object of the paradigm. To achieve this, we have added extensible storage for auxiliary configuration data, called the *registry*, to meta-modeling (and modeling) elements. The registry is a list of registry nodes, each of which has a name and a string value. A registry node can contain further subnodes. The name space of the registry node names is not fixed in order to provide extensibility for external tools. Different tools can store tool-specific information in separate parts of the registry tree.

IV. The Application Model: Prototypes

We have seen that the environment developer fully configures the GME at meta-modeling time by defining the meta-modeling elements. The domain expert then instantiates these meta-objects and creates the application model. Notice that in this type-instance relationship, the domain expert cannot create new types or subtypes of existing meta-modeling elements. To leverage the power of reusability, the GME supports the creation of prototypes and inheritance hierarchies of prototypes in the application model. This greatly increases the productivity of domain experts and the manageability of the application models.

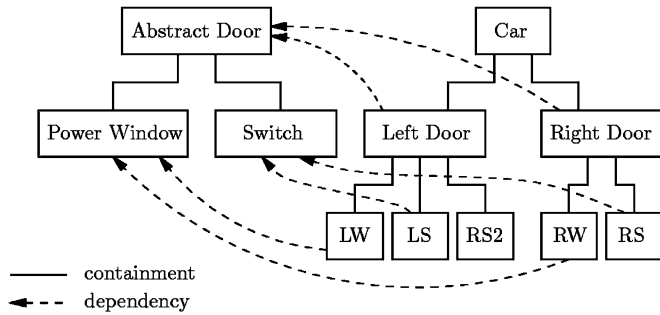

Fig. 8. The cloning process.

A. Prototypes and Clones

A *prototype*[2] is a representative example of a group of objects that can be reused (or *cloned*) at other places in the application model. There is no notion of instantiation, as in class-based OOP, because prototypes exist as independent entities [16]. They are all regular modeling elements, like atoms, references, sets, and composite models. As in other prototype-based systems [23], two mechanisms are provided to construct objects [3]:

1) a mechanism for creating completely new objects;
2) a mechanism for cloning existing objects.

The first mechanism is employed when there is no existing prototype upon which to base the new object. The second mechanism is used to produce a copy of an existing prototype that can be modified to get similar, but not identical, properties. The prototype–clone relationship is preserved for the full lifetime of these objects, which distinguishes cloned objects from simple copies. Cloned objects automatically reflect all changes made in the prototype.

As an example of the cloning process, let us consider a model of a car and its power-window system in Fig. 8. To simplify the figure, we model a coupe with two doors on each side. Each door has a power window and a power-window switch. The left door on the driver side has an additional switch to control the other power window on the RHS. First, we construct an abstract model of a door and then create two clones of it inside the model of the car. Finally, we create a new switch object "RS2" in the "Left Door" clone, which shall control the power window "RW" of the right door. If we decide at some later time to extend the model of the abstract door with additional objects (like a power door lock and its control), then the changes will propagate to both the left and right doors of the car.

In the remainder of this section, we will formalize a few basic requirements (axioms) while examining their consequences. The following two axioms are easy to understand, but they have a profound impact on the rest of the system.

1) An untouched clone, which has never been modified, must behave exactly the same way as its prototype.
2) Each clone has a single prototype.

The second axiom, which in fact prohibits multiple inheritance in the prototype–clone hierarchy, may be argued. In [3], it has been argued that objects tend to be constructed from a single prototype. Although we feel that this is a serious restriction in principle, we have accepted it to ease the implementation and increase the usability[3] of the system. Nevertheless, we have found no logical inconsistencies when clones can have multiple prototypes.

B. The Type of Prototypes and Clones

Recall that each modeling element in the application model has a type, which is the corresponding meta-modeling element. In the previous example, the type of both "Abstract Door" and "Left Door" is "Door," while the type of "Power Window" and "LW" is "Window." Using our first axiom we can see that prototypes and their unmodified clones must have the same type. In fact, this is always the case, as types of modeling elements can never be changed.

The positive side effect of this is that clones have the same set of attributes as their prototypes. By modifying the value of an attribute of some prototype, the change propagates to all clones. As in all prototype-based systems, however, the clones are not identical mirror images of their prototypes. It is possible to overwrite any attribute value in the clone and expect the new value not to be rewritten by the propagation mechanism. This is implemented by an extra flag for each attribute value, which indicates whether the value is inherited from the prototype or explicitly specified in the clone.

C. Composite Clones and the Dependency Hierarchy

Propagation is not limited to the changes of attribute values. By the first axiom, an unmodified clone of some composite model prototype must be indistinguishable from the prototype. Therefore, it must contain clones of the prototype's children, together with their attributes and their children all the way down the containment hierarchy. Again, the clone is not a simple deep copy of the prototype, as the relationship between the prototype and clone is preserved.

There exists a *dependency hierarchy* among the objects of the application model. Each clone depends on its prototype, and the children of each clone depend on the children of the clone's prototype. The dependency relationships between clone and prototype and between their children serve the same purpose, namely, the dependent objects are automatically updated when the object they depend on is modified. Notice that the dependency hierarchy is a disjoint union of up-directed trees (a forest). The dependency hierarchy gets even more interesting when new objects are inserted into a clone. If the inserted object is a newly created one, then the dependency chain stops; otherwise, it is a clone, and the dependency chain continues at some other part of the containment hierarchy. For example, we could add a lock switch to the Left Door in Fig. 8 that could disable to window switch of the right door of our car. This new part could be created in place, that is, it could be a new model object. In this case, there is no upward dependency chain starting here. It would be, however, a better modeling practice to define the lock switch in a different place in the model hierarchy, perhaps as a root object,

[2]See [3, ch. 3] for an excellent introduction to prototype-based languages.

[3]This system is in everyday use in different engineering domains [5], [13], [14]. We have found that some users had difficulty with grasping and productively using even single inheritance.

not even as part of the Abstract Door, and insert a prototype of it in the Right Door. In this case, the dependency chain would continue up to this new root object.

Creating clones of prototypes is a simple operation in the GME—the prototype is selected and then dragged to the destination. This process prevents the introduction of multiple inheritance in the application model. It also differentiates two types of dependencies: an *explicit* dependency is created when the user creates a clone of some prototype, and if the prototype is a model and has children, then the dependencies between the children of the clone and the children of the prototype are *implicit*.

D. References, Sets, and Propagation

The dependency relationship is not limited to models, atoms, and attributes. It carries over to connections, references, and sets, as well. However, connections cannot be retargeted; only their attributes can be modified in clones. On the other hand, reference and set relations can be modified in clones. The propagation of sets and references has the same semantics as that of attributes. In the case of sets, either the entire selection is propagated from the prototype or it is completely overwritten in the clone. The attribute value propagation behaves the same way in all FCOs.

The set and reference selections, and the attribute values, propagate down the dependency hierarchy. The propagation of a particular value can be broken at any node by setting the flag and explicitly specifying the new value.

E. The Consequence of the Second Axiom

The seamless integration of the prototype and containment hierarchies presents a unique challenge. Observe that inheritance and containment of classes in OOP are not as complex as here. In OOP, composite classes can be derived, but only the derived class itself can be enhanced. None of the contained (locally defined) classes inherited from the base class are enhanced. In fact, in OOP, contained classes are not carried over into the derived class at all; they are merely accessible.

According to the rules we have presented so far, it is possible to violate the second axiom by implicit dependencies. Consider the situation in Fig. 9, where model A contains another model B and a clone C of B. We then create a clone A' of model A, which in turn contains models B' and C' cloned from B and C, respectively. Clearly, C depends on B, A' on A, B' on B and C' on C. Now we expect that A' behaves the same way as A does until we modify it. This means that C' must be cloned from B' as well, showing that C' has two prototypes. This clearly contradicts our second axiom. To avoid this situation, we have chosen the somewhat limiting solution that the (explicitly cloneable) prototypes must be root objects contained in folders. This makes C impossible to clone from B. While this restriction is not elegant from a purely scientific point of view, it simplifies the environment, both its implementation and its use.

Notice that a weaker restriction would have been sufficient to prevent this situation, namely, the requirement that no prototype A can contain another prototype B and its clone C. This would prohibit embedded dependencies in prototypes. Tto enforce this restriction, however, we would need to disallow the creation of the clone C of B if A has clones already and allow this operation

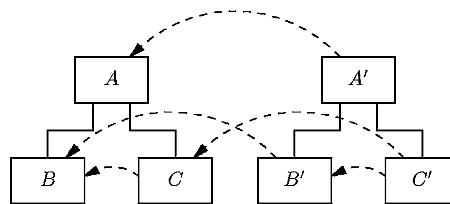

Fig. 9. Multiple inheritance in the prototype–clone hierarchy at C'.

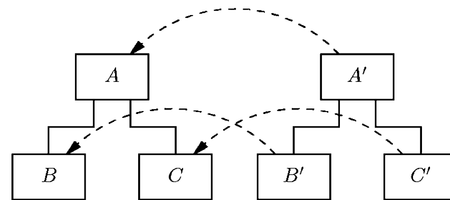

Fig. 10. Clone of a composite object.

if A has no clones. This would differentiate regular objects and prototype objects (objects that have clones) in a fundamental way.

Our approach simplifies the handling of relations other than the containment hierarchy, as well Fig. 10. Consider the following example where model A contains atoms B and C, and the clone A' of A contains the clones B' and C' of B and C, respectively. Imagine that a user would like to create a connection between B and C in A. Clearly, the framework has to create a cloned connection in A' and needs to identify the endpoints of the cloned connection in A'. Because of our restriction on the explicit derivations, atom B cannot have clones in A' other than the implicit clone B', which makes this identification trivial.

F. Deletion of Objects

We now return to the first axiom to help us understand the behavior of a consistent implementation when objects are deleted. Consider Fig. 9 again and imagine that a user would like to delete B. Since A' must be an identical copy of A, as long as it is not modified, the system has to delete the clone B' of B. This shows that all clones of a prototype must be automatically deleted whenever the prototype itself is deleted.

Having dealt with the deletion of prototypes, we turn our attention to that of clones. Clearly, it must be possible to delete an explicitly derived clone, such as A' in our example. The issue is whether one can delete the implicitly derived clones, such as B' and C'. We have forbidden this for the following reasons. First, this operation would be impossible to undo manually, using regular editing operations. This is because of the restriction that explicit prototypes, the atom B in our case, must be root objects contained in folders. Second, this behavior is analogous to that of the class inheritance in OOP. One cannot delete contained classes, functions, etc., in derived classes; they can only be hidden to some extent.

V. Meta-Model Composition

Just as the reusability of domain models from application to application is essential, the reusability of meta-models from domain to domain is also important. Ideally, a library

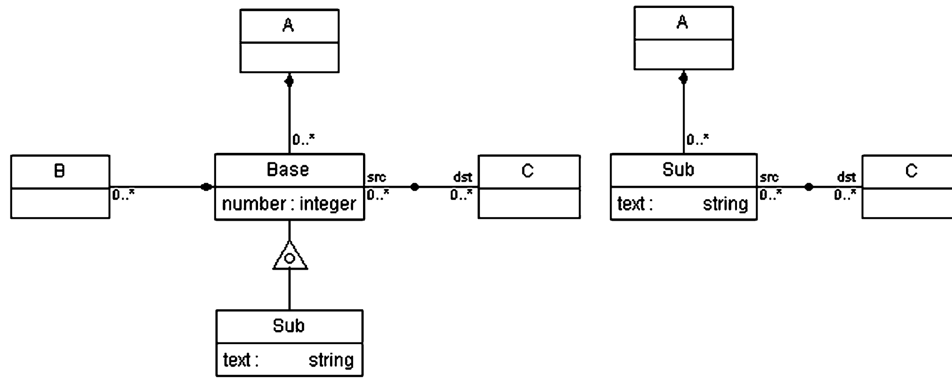

Fig. 11. Interface inheritance.

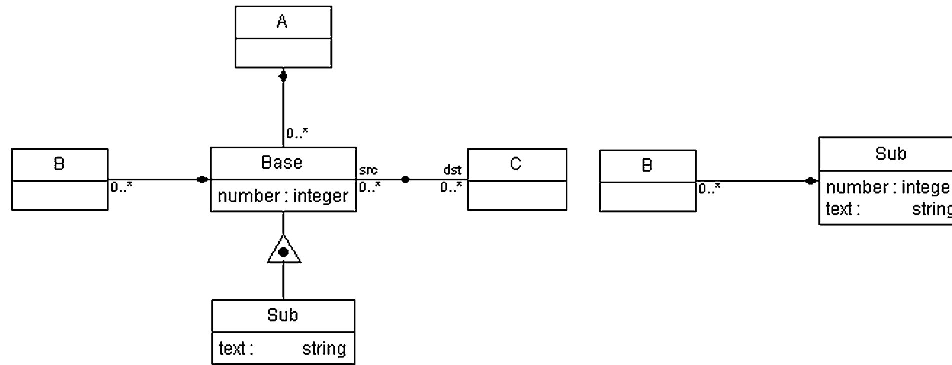

Fig. 12. Implementation inheritance.

of meta-models of important subdomains should be made available to the meta-modeler, who can extend and compose them together to specify multi-paradigm domain languages. These subdomains 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 meta-models, just as subclasses do not modify base classes in OOP. Changes in the meta-model libraries, reflecting a better understanding of the given domain, for example, can then propagate automatically to the meta-models 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 meta-modeling language is based on UML class diagrams. However, to support meta-model composition, some new "operators" were necessary. These operators are new graphical symbols that extend the visual syntax used in UML class diagrams. The operators are used to compose elements from two class diagrams to form the content of a new class diagram. Note that the new class diagram is implicitly represented by specifying its constituent diagrams and how they are composed.

The union operator is used to represent the direct composition 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. The union can be thought of as defining the "join points" or "composition points" of two or more source meta-models. Notice that multiple inheritance can be used instead of union. For example, the union of two classes A and B is equivalent to introducing a new class AB that is inherit from both A and B. The only difference is that in the latter case, all three classes A, B, and AB exist in the composed language, while applying the union operator results in a single composite class.

New operators were also introduced to provide finer control over inheritance. While it is usually not a good practice to extend existing well-known languages, such as UML, supporting meta-model composition and the requirement that no modification be made to existing meta-models mandated this extension. 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. In the sample meta-model in the left-hand side (LHS) of Fig. 11 Sub is derived from Base using interface inheritance. The RHS shows the equivalent class diagram when inheritance is not used. Notice that A can contain Sub objects, and there is an association between C objects and Sub-s. But the number attribute of Base is not present in Sub, and Sub cannot contain B objects.

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. Fig. 12 shows the same class diagram as Fig. 11, except that now implementation inheritance is used to derive Sub from Base. The RHS shows again how this would need to be modeled without

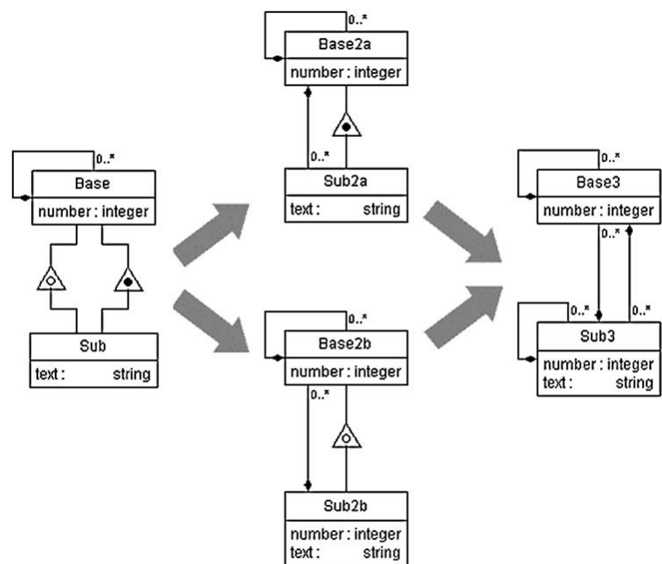

Fig. 13. Implementation and interface inheritance.

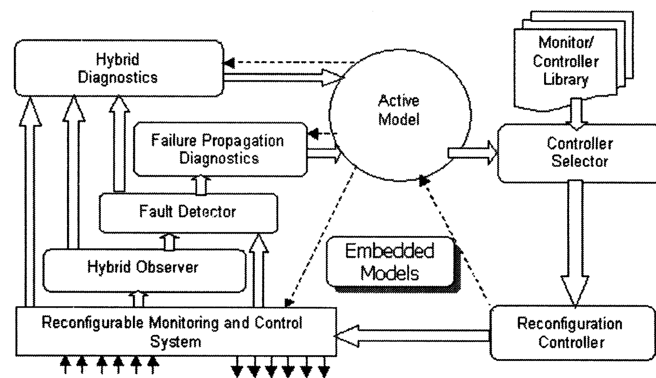

Fig. 14. Fault-Adaptive Control Architecture.

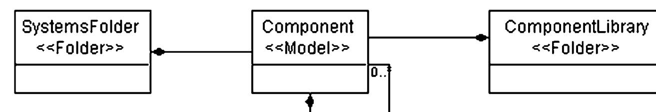

Fig. 15. Top-level meta-model for plant modeling.

inheritance. Subs can contains B-s, and it has the number attribute. There is no association between C-s and Sub-s, and A-s cannot contain Sub-s either (refer to the example in Section I for another demonstration of this operator).

Notice that the union of these two new inheritance operators is the "regular" UML inheritance. Consider a simple class diagram where a class Base can contain instances of Base and has a subclass Sub. An equivalent class diagram without inheritance would look like the RHS of Fig. 13. Here instances of Base3 can contain instances of Base3 and Sub3, and, similarly, instances of Sub3 can contain instances of Sub3 and Base3. We get the same result if we use both interface and implementation inheritance between Base and Sub regardless of the order they are applied, as illustrated in the figure.

Consider the LHS of Fig. 13. 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.

It is important to observe that the use the union 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 meta-model could be represented without the new operators. However, such meta-models would either need to modify the original meta-models or require the manual repetition of information in them due to the lack of fine control over inheritance. These meta-models would also be significantly more cluttered, making the diagrams more difficult to read and understand.

VI. An Example for Meta-Modeling: The Fault-Adaptive Control Technology Project

The Fault-Adaptive Control Technology (FACT) project [27] aims at developing technology for complex control applications, where faults in sensors, actuators, and the plant are anticipated, and to address these problems the issues of controller reconfiguration are systematically explored. The goal is to provide tools for the designer to build fault-adaptive control systems that can survive faults. The proposed generic architecture is based on integrating high-fidelity diagnostics of dynamic systems with control technology, where the control portion can be influenced by the results of the diagnosis. The generic architecture is instantiated using a model-based approach: the designer builds models of the plant and the corresponding control system, and software generators synthesize the running application: a problem-specific instance of the generic architecture.

The overall approach, illustrated in Fig. 14, is using model-based techniques for hybrid observation, for the detection, isolation, and estimation of faults, and for controller selection. We assume the systems we deal with combine the continuous and discrete (switching) dynamics of the plant with regulators operating on sampled data and discrete-event supervisory controllers. Hybrid models [28], derived from hybrid bond graphs [29], allow the systematic modeling of the continuous and discrete dynamics of the plant. The supervisory controller, modeled as a finite-state automaton, generates the discrete events that cause reconfigurations in the plant. Fault detection involves comparison of the expected behavior of the plant (generated from the hybrid models) with actual system behavior, to determine when discrepancies occur. This requires the design and implementation of hybrid observers that estimate the continuous dynamic states of the system and detect mode changes during system operation. Sophisticated signal analysis and filtering methods linked to the hybrid observers are used for detecting deviations from nominal behavior and triggering the fault-isolation schemes.

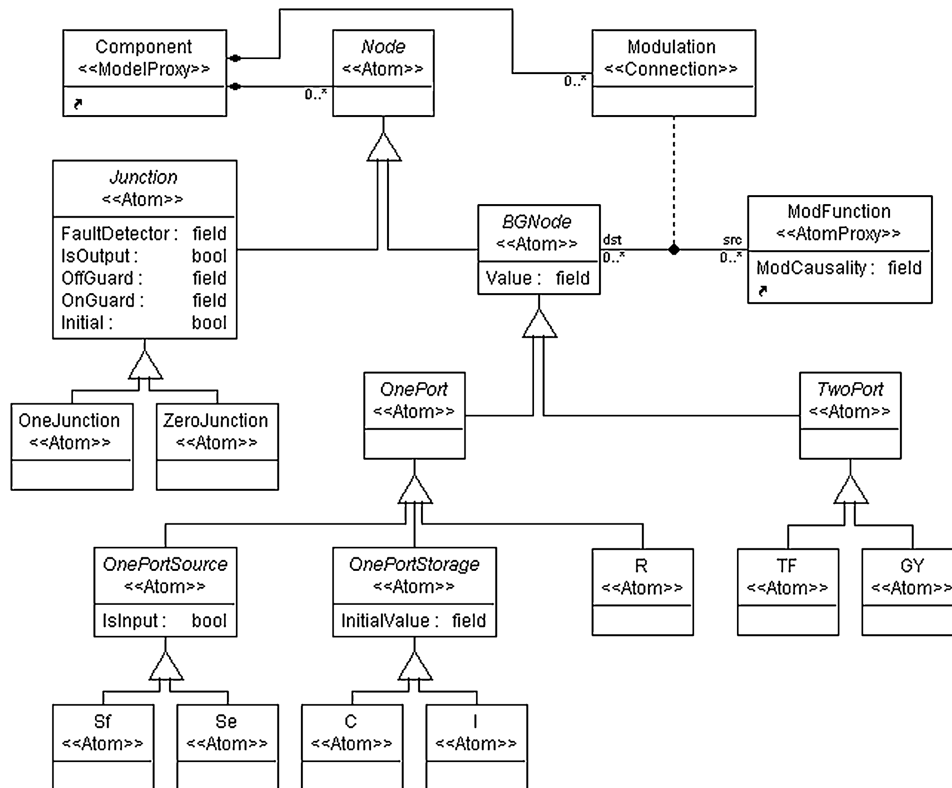

Fig. 16. Meta-model for bond-graph nodes.

Our diagnostic schemes integrate the use of failure-propagation graph-based techniques for discrete-event diagnosis [5] and combined qualitative reasoning and quantitative parameter estimation methods for computationally efficient fault isolation [31] of degraded components (sensors, actuators, and plant components). The dynamic state accumulated from the observer (discrete system mode plus continuous state vector) and fault-isolation units (status of faulty and degraded sensors, actuators, and plant components) define the active system state model. The tracking, fault detection, and fault-isolation mechanisms, illustrated on the LHS of Fig. 17, together constitute a bottom-up computational approach for estimating the dynamic system state (nominal or faulty) by monitoring plant and controller variables.

When faults are detected, the diagnostics component executes a fault-isolation procedure and updates the active (state) model accordingly. The reconfiguration controller uses this information to select from the controller library the controller that is most effective in maintaining desired system operation and performance. The selection and reconfiguration mechanisms operate in a top-down manner, using the dynamic-state information to effect changes in supervisory control mechanisms, selection (not synthesis) of feedback control mechanisms, and retuning of low-level regulators, such as Proportional-Integral-Derivative (PID) or model-based controllers. The overall computational architecture combines the bottom-up and top-down computational schemes in a seamless manner, via the shared active model.

As mentioned previously, the generic architecture is instantiated for a particular application from a set of models that were built using a modeling environment. The modeling language used was defined by the meta-models as follows. The modeling paradigm has been decomposed into the following two major ingredients:

1) the modeling language for representing physical components;
2) the modeling language for representing controllers.

Hereafter, we will we show the core concepts in both kinds of meta-models.

A. Meta-Models for Plant Modeling

On the highest level, the meta-model declares that our model database will contain physical Components, which might be contained in ComponentLibrary folders, as shown on Fig. 15.

A Component, in this context, corresponds to a physical entity in the plant (e.g., a valve, a pipe, and a motor), an instance of which may be part of an energy-based bond-graph model. These graphs consist of bond-graph nodes, as the meta-model fragment on Fig. 16 shows.

One can recognize the familiar concepts from bond-graph modeling: 1- and 0-junctions; R, C, and I elements; and flow and effort sources. Bond graphs also need connections whose meta-model is shown in Fig. 17.

In this language, components have energy ports, which are connected to (internal) junctions and one- or two-port elements. These ports are used on one level above the components to model energy flows across and between components.

The bond-graph-based detailed modeling is complemented by another modeling aspect where discrete failure modes and

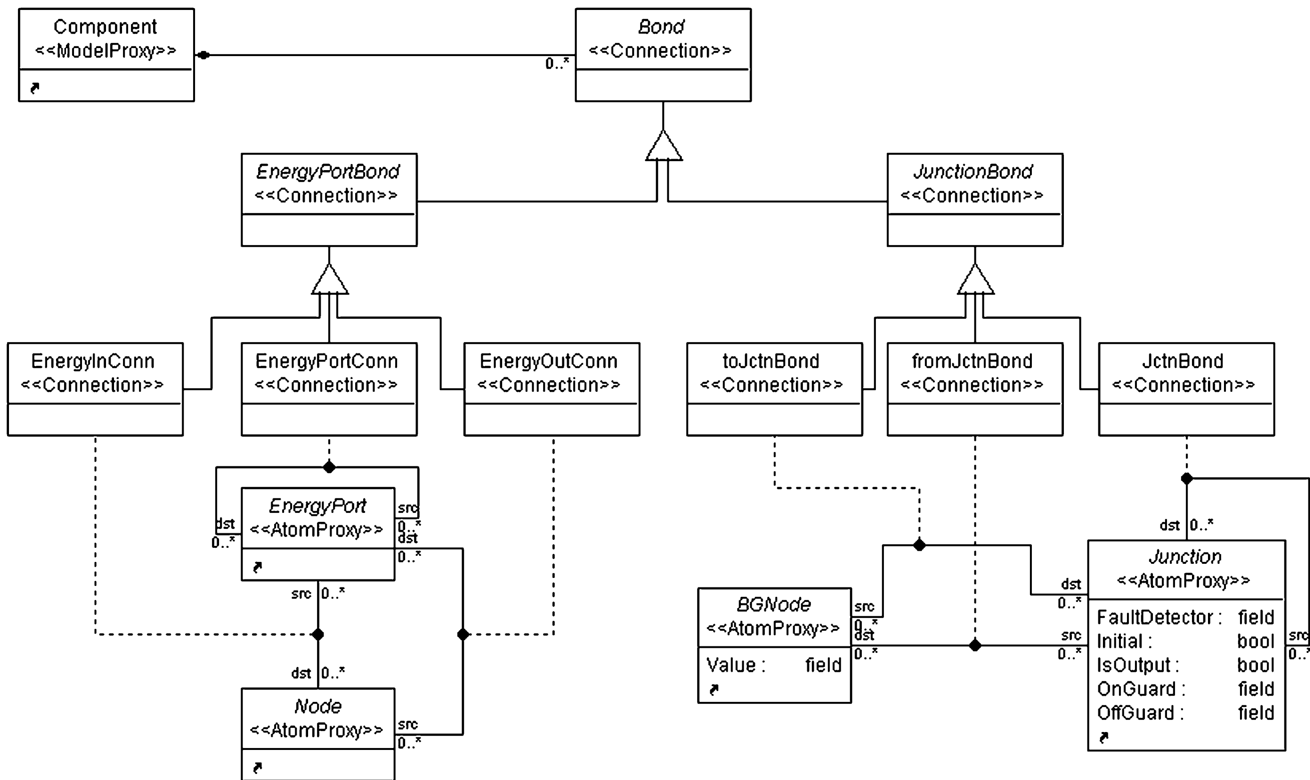

Fig. 17. Meta-model for bond-graph connections.

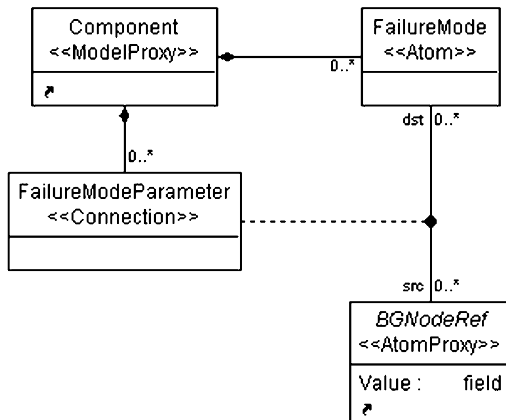

Fig. 18. Component failure modes.

their functional effects are modeled. Physical components can have (atomic) failure modes, as shown on Fig. 18.

Failures can introduce cascades of functional failures, called discrepancies. These cascades form failure-propagation graphs, which have a meta-model shown on Fig. 19.

B. Meta-Models for Controller Modeling

The controllers in the FACT domain are implemented as software components running on a platform called Open Control Platform (OCP), which provides an integration platform and run-time infrastructure [33], based on the concepts of real-time CORBA. OCP supports a specific (software) component model, where components encapsulate control algorithms operating on packets of data, which are passed from component to component via a dataflow network. The modeling language used for controller modeling was tailored to support this approach, and its meta-model is shown in Fig. 20.

Controllers are represented as networks of OCPComponents that implement the control algorithms. OCPComponents operate on typed packets of data, and the typing is specified via OCPSignals, which are collections of attributes (like fields of a structure in C). The control algorithms are represented inside of components in the form of OCPBehaviors. An OCPBehavior can be implemented as code in C++ or in the form of a finite-state machine (from which C++ code is generated). Each component has a fixed number of input and output ports through which they receive and send data. The ports carry signals, which have an associated data type in the form of an OCPSignal. The components are scheduled for execution when they receive data on their input ports. Once the data is processed, they send the data produced through their output ports. When the behavior is specified with the help of a finite-state machine, input data trigger might trigger a transition from the current state to a next state, and data-producing computation can be associated with the transition or with a state.

OCPComponents are embedded in OCPSystems, which are containers specifying system-level properties (OCPModes, etc.) that are required in the run-time system. Once the systems containing one or more controllers are modeled and the executable code for the behaviors is supplied, a model interpreter tool translates the models into configuration files (for tools like Visual Studio) and C++ source code (containing "glue" code for configuring the objects). The designer can focus on the details of the

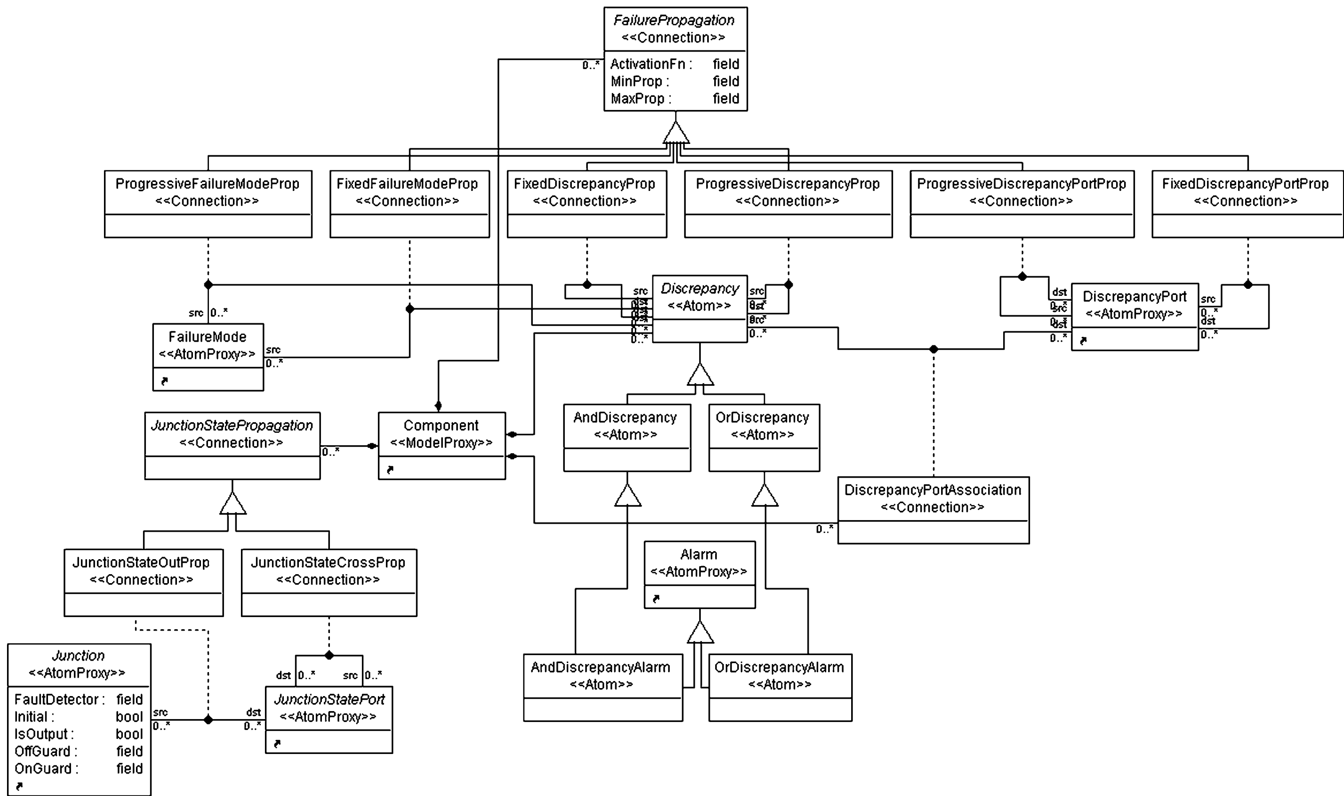

Fig. 19. Meta-model for failure propagations.

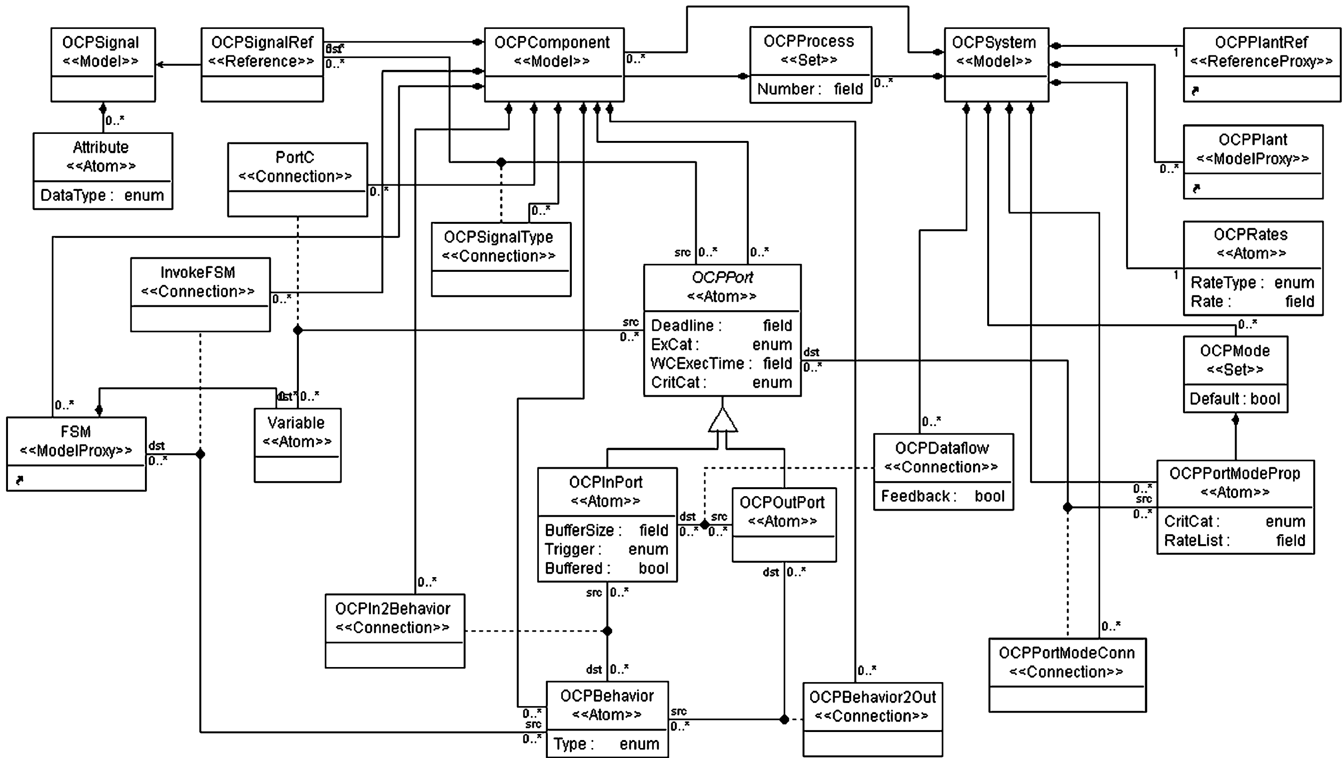

Fig. 20. Meta-model for controller models.

control algorithms and just supply a high-level (visual) specification of the controller architecture, as the software generator tool will produce all the "housekeeping" code automatically, from the models.

VII. Conclusion

Tools that are used to support the creation of complex multi-paradigm domain-specific models must provide numerous ca-

pabilities to handle complexity and manageability. The tool developers must further consider the different roles of the users involved and provide complexity-managing capabilities for each group. In some respects, the tool developers face problems that are similar to those difficulties experienced by programming language designers.

A study of the history of programming languages reveals the great benefit realized from the introduction of typing facilities [25]. The programmer's ability to create their own user-defined types offers the advantage of being able to generalize and describe the key properties of a common set of entities from the problem domain. Of course, the ability to categorize and use global names to refer to types is not only an advantage for writing programs. Types and prototypes are two capabilities that can be very useful in modeling. Modeling tools that support these concepts provide mechanisms to share a common description among numerous objects.

In our modeling tool, the GME, types, and prototypes allow the modeler to categorize and manage common modeling concepts. A prototype represents the default description for a modeling concept. Cloning a particular prototype can create new objects. They reuse part of the knowledge stored in the prototype by saying how the new object differs from the prototype.

Although the literature on types and prototypes has been cited throughout the paper with respect to programming languages, there appears to be little research into the use of types in domain-specific modeling environments. An application of types within a modeling environment is presented in [15]. This work focuses on the desire to perform type checking on the dynamic interactions between component descriptions. Their implementation of type checking involves automata and reflection. The intention of incorporating types into the GME, however, has reuse of prototypical instantiation as a primary goal.

There are several challenging opportunities to extend these ideas. A natural target for further research is to explore the intriguing interaction of the prototype–clone and containment hierarchies and the connection, reference and sets relations. Particularly interesting topics are the systematic study of multiple inheritance in the prototype–clone hierarchy and the introduction of a hiding mechanism in clones (to simulate the deletion of implicit clones).

REFERENCES

[1] B. Abbott, T. Bapty, C. Biegl, G. Karsai, and J. Sztipanovits, "Model-based approach for software synthesis," *IEEE Softw.*, vol. 10, pp. 42–54, May 1993.

[2] G. Booch, I. Jacobson, and J. Rumbaugh, *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1998.

[3] I. Craig, *The Interpretation of Object-Oriented Programming Languages*. Berlin: Springer-Verlag, 2000.

[4] C. J. Date, *An Introduction to Database Systems*, 8th ed. Reading, MA: Addison-Wesley, 2003.

[5] J. Davis, J. Scott, J. Sztipanovits, and M. Martinez, "Multi-domain surety modeling and analysis for high assurance systems," in *Proc. Engineering Computer Based Systems (ECBS)*, Nashville, TN, Mar. 1999, pp. 254–260.

[6] J. Gray, T. Bapty, S. Neema, and J. Tuck, "Handling crosscutting constraints in domain-specific modeling," *Commun. ACM*, pp. 87–93, Oct. 2001.

[7] R. Hilliard, "Views and viewpoints in software systems architecture," in *Proc. First Working IFIP Conf. Software Architecture*, San Antonio, TX, Feb. 22–24, 1999.

[8] *Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE Standard 1471-2000, Oct. 2000.

[9] G. Karsai, "A configurable visual programming environment: A tool for domain-specific programming," *IEEE Computer*, vol. 28, pp. 36–44, Mar. 1995.

[10] G. Karsai, G. Nordstrom, A. Ledeczi, and J. Sztipanovits, "Specifying graphical modeling systems using constraint-based metamodels," presented at the IEEE Symp. Computer Aided Control System Design, Anchorage, AK, Sept. 25, 2000.

[11] S. Kelly and J. Tolvanen, "Benefits and experiences of visual domain-specific modeling with metaCASE," presented at the Int. Workshop Model Engineering—ECOOP 2000, Nice, France, June 13, 2000.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting started with AspectJ," *Commun. ACM*, pp. 59–65, Oct. 2001.

[13] A. Ledeczi, M. Maroti, G. Karsai, and G. Nordstrom, "Metaprogrammable toolkit for model-integrated computing," in *Proc. Engineering Computer Based Systems (ECBS)*, Nashville, TN, Mar. 1999, pp. 311–317.

[14] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing domain-specific design environments," *IEEE Computer*, pp. 44–51, Nov. 2001.

[15] E. A. Lee and Y. Xiong, "System-level types for component-based design," in *Proc. 1st Int. Workshop Embedded Software*, Tahoe City, CA, Oct. 2001, pp. 237–253.

[16] H. Lieberman, "Using prototypical objects to implement shared behavior in object oriented systems," in *Proc. Object-Oriented Programming Systems, Languages, Applications (OOPSLA)*, Portland, OR, Nov. 1986, pp. 214–223.

[17] E. Long, A. Misra, and J. Sztipanovits, "Increasing productivity at Saturn," *IEEE Computer*, vol. 31, pp. 35–43, Aug. 1998.

[18] M. Moore, S. Monemi, J. Wang, J. Marble, and S. Jones, "Diagnostics and integration in electrical utilities," in *Proc. IEEE Rural Electric Power Conf.*, Louisville, KY, May 2000, pp. C21–C210.

[19] G. Nordstrom, J. Sztipanovits, G. Karsai, and A. Ledeczi, "Metamodeling—Rapid design and evolution of domain-specific modeling environments," in *Proc. Engineering Computer Based Systems (ECBS)*, Nashville, TN, Mar. 1999, pp. 68–74.

[20] B. Nuseibeh, J. Kramer, and A. Finkelstein, "A framework for expressing the relationship between multiple views in requirements specification," *IEEE Trans. Software Eng.*, vol. 20, pp. 760–774, Oct. 1994.

[21] J. Sztipanovits, G. Karsai, and T. Bapty, "Self-adaptive software for signal processing," *Commun. ACM*, pp. 66–74, May 1998.

[22] J. Sztipanovits, G. Karsai, and H. Franke, "Model-integrated program synthesis environment," in *Proc. Engineering Computer Based Systems (ECBS)*, Friedrichshafen, Germany, Mar. 11, 2000, pp. 348–355.

[23] D. Ungar and R. Smith, "Self: The power of simplicity," in *Proc. Object-Oriented Programming Systems, Languages Applications (OOPSLA)*, Orlando, FL, Dec. 1987, pp. 227–242.

[24] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling With UML*. Reading, MA: Addison-Wesley, 1999.

[25] P. Wegner, "Programming languages—The first 25 years," *IEEE Trans. Comput.*, pp. 1207–1225, Dec. 1976.

[26] The Extensible Markup Language (XML) [Online]. Available: http://www.w3c.org/XML/

[27] Fault-Adaptive Control Technology (FACT) Project [Online]. Available: http://www.isis.vanderbilt.edu/Projects/Fact/Fact.htm

[28] M. S. Branicky, V. Borkar, and S. Mitter, "A unified framework for hybrid control: Background, model, and theory," in *Proc. 33rd IEEE Conf. Decision Control*, Paper No. LIDS-P-2239, Lake Buena Vista, FL.

[29] P. J. Mosterman and G. Biswas, "A theory of discontinuities in physical system models," *J. Franklin Inst.*, vol. 335B, pp. 401–439.

[30] A. Misra, J. Sztipanovits, and J. Carnes, "Robust diagnostics: Structural redundancy approach," in *Proc. Knowledge Based Artificial Intelligence Systems Aerospace Industry, SPIE's Symp. Intelligent Systems*, Orlando, FL, Apr. 5–6.

[31] E. J. Manders, S. Narasimhan, G. Biswas, and P. J. Mosterman, "A combined qualitative/quantitative approach for efficient fault isolation in complex dynamic systems," in *Proc. 4th Symp. Fault Detection, Supervision Safety Processes*, pp. 512–517.

[32] M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*, D. Schmidt, Ed. New York: Wiley, 2000.

[33] J. Paunicka, B. Mendel, and D. Corman, "The OCP—An open middleware solution for embedded systems," in *Proc. American Control Conf.*, 2001.

Gabor Karsai received the B.Sc., M.Sc., and Technical Doctorate degrees from the Technical University of Budapest, Budapest, Hungary, in 1982, 1984, and 1988, respectively, and the Ph.D. degree from Vanderbilt University, Nashville, TN, in 1988.

He is an Associate Professor of Electrical and Computer Engineering at Vanderbilt University and Technical Director of the Institute for Software-Integrated Systems (ISIS) at Vanderbilt. He conducts research in model-integrated computing (MIC), in the field of development environments, automatic program synthesis, and the application of MIC in various government and industrial projects. Since 1988, he has been involved in various projects on fault diagnostics modeling and algorithms, which have been applied and demonstrated in various embedded systems, including chemical manufacturing plants, the Space Station, and next-generation fighter aircraft. Currently, he is Principal Investigator on various Defense Advanced Research Projects Agency (DARPA)-sponsored research projects on fault-adaptive control systems, autonomic logistics, and software synthesis for embedded systems.

Dr. Karsai is a Member of the IEEE Computer Society and the Technical Committee on Computer-Based Systems.

Miklos Maroti received the Ph.D. degree in mathematics from Vanderbilt University, Nashville, TN, in 2002.

He is a Research Assistant Professor at the Institute of Software Integrated Systems (ISIS), Vanderbilt University. His current research interest includes formal specification and analysis of embedded systems and active libraries of middleware components.

Akos Ledeczi received the M.Sc. degree in electrical engineering from the Technical University of Budapest, Budapest, Hungary, in 1989 and the Ph.D. degree in electrical engineering from Vanderbilt University, Nashville, TN, in 1995.

He is a Senior Research Scientist at the Institute for Software Integrated Systems (ISIS), Vanderbilt University. His current research interests include tools for visual modeling of complex systems, model-based synthesis, and simulation of embedded systems and distributed systems.

Jeff Gray (S'88–M'02) received the B.S. and M.S. degrees in computer science from West Virginia University, Morgantown, WV, in 1991 and 1993, respectively, and the Ph.D. in computer science from Vanderbilt University, Nashville, TN, in 2002.

He is an Assistant Professor in the Computer and Information Sciences Department at the University of Alabama at Birmingham. His research interests are aspect-oriented software development, generative programming, and model-integrated computing.

Janos Sztipanovits (M'86–SM'90–F'01) graduated from the Technical University of Budapest, Budapest, Hungary, in 1970. He received the "Candidate of Technical Sciences" degree from the Hungarian Academy of Sciences, Budapest, in 1980 and the distinguished doctor degree (Golden Ring of the Republic) in 1982.

Between 1999 and 2001, he worked as Program Manager and Acting Deputy Director of the Defense Advanced Research Projects Agency (DARPA) Information Technology Office. As Program Manager, he worked on the Autonomous Negotiating Teams (ANTs), Model-Based Integration of Embedded Software (MoBIES), and Networked Embedded Software Technology (NEST) programs. He is currently E. Bronson Ingram Distinguished Professor of Engineering in the Electrical Engineering and Computer Science Department of Vanderbilt University, Nashville, TN. He is founding Director of the Institute for Software Integrated Systems (ISIS), Vanderbilt University. During the past two decades, he has conducted research on model-integrated computing (MIC), structurally adaptive systems, and embedded software and systems. He has published more than 150 papers, and he is the coauthor of two books.