Domain Independent Generative Modeling

Branislav Kusy[1], Ákos Lédeczi[1], Miklós Maróti[1], Péter Völgyesi[2]
[1]*Institute for Software Integrated Systems,*
Vanderbilt University, Nashville, TN, USA
{branislav.kusy, akos.ledeczi, miklos.maroti}@vanderbilt.edu
http://www.isis.vanderbilt.edu
[2]*Embedded Information Technology Research Group – Hungarian Academy of Sciences*
Budapest University of Technology and Economics, Budapest, Hungary
volgyesi@mit.bme.hu

Abstract

Model Integrated Computing employs domain-specific modeling languages for the design of Computer Based Systems and automatically generates their implementation. These system models are declarative in nature. However, for complex systems with regular structure, as well as for adaptive systems, a more algorithmic approach is better suited. Generative modeling employs architectural parameters and generator scripts to specify model structure. This paper describes an approach that enables the addition of generative modeling capabilities to any domain-specific modeling language using metamodel composition. The approach is illustrated through an image processing application using the Generic Modeling Environment (GME).

1. Introduction

Modeling and automatic code generation is the most promising way to increase software productivity. One approach employs a single modeling language developed specifically for software modeling to capture the important characteristics of the system under development and then generate a portion of the implementation automatically. The most prominent representative of this technique is the UML. Another approach advocates the use of domain-specific modeling languages, where the language is tailored for the unique needs of the target application domain. This approach enables the modeling of not only the software, but all other aspects of the target system, including its hardware, environment and their relationship.

Model-Integrated Computing (MIC) is a representative of the latter approach. Both techniques need tool support to enable their practical application. If the language is fixed, in case of the UML for example, a single toolset is sufficient. However, creating domain-specific visual model building, constraint management, and automatic program synthesis components for each new domain would be cost-prohibitive for most domains. The Generic Modeling Environment (GME) is a configurable environment that makes it possible to create highly domain-specific environments rapidly.

The configuration is accomplished through UML and OCL-based [3,4] metamodels that specify the modeling paradigm (modeling language) of the application domain. The modeling paradigm contains all syntactic, semantic, and presentation information regarding the domain; which concepts will be used to construct models, what relationships may exist among these concepts, how the concepts may be organized and viewed by the modeler, and what rules govern the construction of the models. The modeling paradigm defines the family of models that can be created using the resulting modeling environment.

The metamodels specifying the modeling paradigm are used to automatically generate the target domain-specific environment. The generated domain-specific environment is then used to build domain models that are stored in a model database. These models are used to automatically generate the applications or to synthesize input to different COTS analysis tools. GME has an open component-based architecture. It allows access to metamodels, models and model modification events through a set of Microsoft COM interfaces.

Just like most software models, models captured in GME are declarative. They describe a particular solution to a particular problem in the given engineering domain in a declarative manner. While this works very well most of the time, there are two cases where a more flexible approach is called for. When the models are large and have a regular, repetitive structure it is more natural to capture the information in an algorithmic manner. Similarly, adaptive systems are cumbersome, if not impossible to capture in a declarative way.

One solution is to represent these architectures in a generative manner. Here, the components of the architecture are prepared, but their number and connectivity patterns are not fully specified. Instead, a generative description is provided that specify how the architecture should be generated "on-the-fly." A generative architecture specification is similar to the generate statement used in VHDL: it is essentially a program that, when executed, generates an architecture by instantiating components and connecting them together.

The generative description is especially powerful when it is combined with architectural parameters and hierarchical decomposition. In a component one can generatively represent architecture, and the generation "algorithm" can receive architectural parameters from the current or higher levels of the hierarchy. These parameters influence the architectural choices made (e.g. how many components to use, how they are connected, etc.), but they might also be propagated downward in the hierarchy to components at lower levels. There this process is repeated: architectural choices are made, components are instantiated and connected, and possibly newly calculated parameters are passed down further. Thus, with very few generative constructs one can represent a wide variety of architectures that would be very hard, if not impossible, to pre-enumerate.

Our earlier work in self-adaptive systems demonstrated the power of generative modeling in one particular domain [5]. This paper describes the generalization of the approach. In particular, we have created a generative representation methodology that can be easily added to any domain-specific modeling language. The approach is based on metamodel composition. Furthermore, we have developed a combination of a code generator and a model interpreter that executes the generator scripts according to the current instantiation of the architectural parameters and creates the corresponding pure declarative model. The approach is detailed in the following sections.

2. Generative Modeling

Let us illustrate generative modeling through an example. Data parallelism is often utilized in image processing. Consider, for example, the signal flow of a convolution running on four processors as shown in Figure 1. The input image is split four ways, the convolution is carried out in parallel on the subimages and then the results are merged to form a single image. If the number of available processors changes, the signal flow model needs to be modified to match it. However, the structure is quite regular, it is natural to express it in an algorithmic manner as a function of the number of processors.

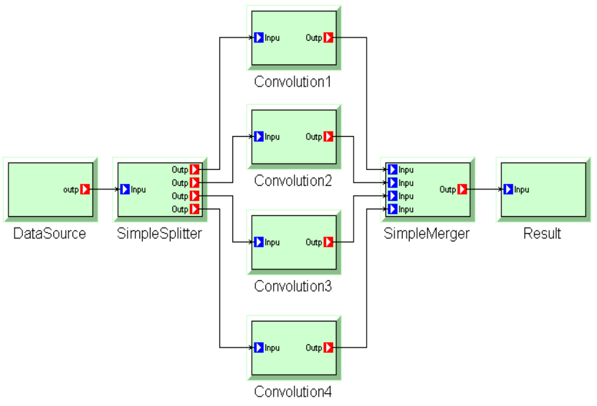

Figure 1. Data Parallel Convolution

Consider Figure 2. It shows the same components as Figure 1, but the actual signal flow is missing. Instead, the components are connected to a generator block called SplitAndMerge. There is an architectural parameter connected to it also (called NumProcessors) that have a numerical attribute (not shown). Generators have a textual attribute containing the generator script that describes the desired model structure utilizing the architectural parameters, in this case NumProcessors.

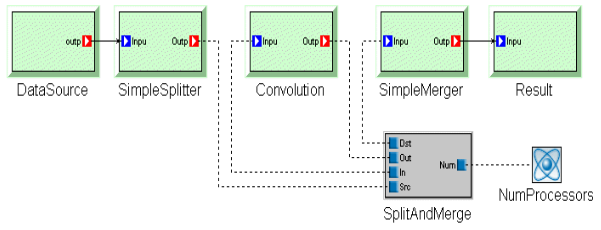

Figure 2. Generative Model

Executing the generator creates the required signal flow model. When the number of processors changes, the value of NumProcessors is the only thing that needs to change. This is certainly easier and less error-prone than manually redrawing the models to reflect the change. In the remainder of this section we describe how this is actually accomplished emphasizing how generative modeling can be easily added to any domain-specific language using our technology.

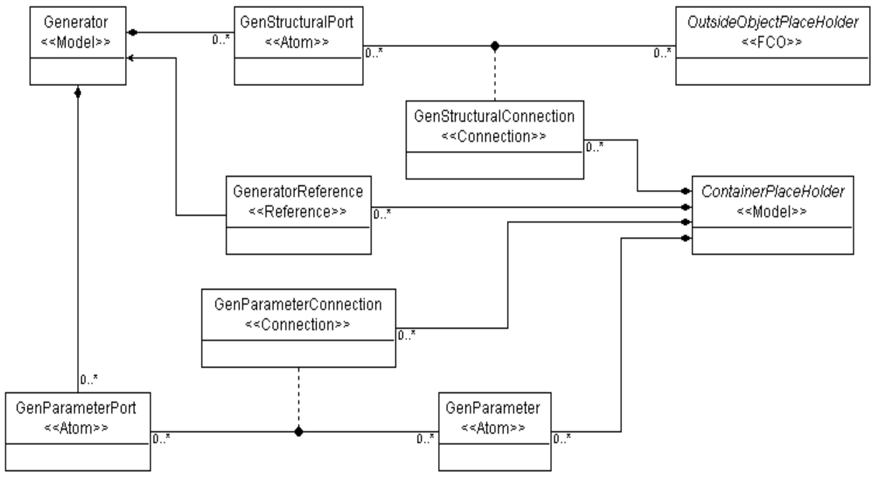

Figure 3. Generative Metamodel

2.1. Generative metamodel

Model Integrated Computing employs metamodels to define domain-specific modeling languages [1]. Metamodel composition is utilized to allow the combination of existing languages to form more complex ones. Metamodel composition simply means taking two or more metamodels and specifying relations among some of their modeling concepts [6]. These relations define how models in the original languages can be composed together in the new language.

Figure 3 shows the metamodel specifying generative modeling. It is only a partial metamodel, i.e. it does not define a meaningful modeling language in and of itself. It was specifically created, so that it can be composed with any other metamodel to add generative modeling capability to the corresponding modeling language.

The key concept is the use of generic objects acting as placeholders - objects that are not completely defined in the generative metamodel. These objects will be fully defined only after metamodel composition, i.e. after the user decides how she wants to enable generative modeling in the target domain-specific language. Even though the placeholders are not fully defined, we can specify all of their properties related to generative modeling. Furthermore, there are several modeling concepts that are completely specified in the generative metamodel; these are specific to generative modeling and do not need to be composed.

Consider Figure 3. The main object from the generative point of view is the Generator model containing GenStructuralPorts and GenParameterPorts. These ports can be connected to objects from the user defined metamodel (to be composed) and to architectural parameters (GenParameter) respectively. The Generator model has the GeneratorScript textual attribute that will contain the algorithmic specification of the desired model structure.

The abstract object OutsideObjectPlaceholder is introduced to define the connectivity between generators and objects from the user defined metamodel through GenStructuralConnections. Containment is modeled using the ContainerPlaceholder model. This is where generative models can be contained in the final target environment.

An important concept for the reusability of generators is the GeneratorReference. As you can see in Figure 3, Generators are not contained in any other models; they are standalone. In fact, a generator script can only refer to its own ports (structural and parameter) and has no knowledge of the “outside world,” where and how it is being used (for a more detailed description, see section 4). Generators capture structural information only, so a SplitAndMerge generator model can be used without modification in any domain-specific language from signal flow through ADLs to state machines. Therefore, it is not Generator models that are inserted into domain-specific models, but references to Generators. (References are just like pointers in programming languages.) This enables using the same Generator model in different models using different architectural parameters.

For example, the SplitAndMerge object in Figure 2 is a reference to a SplitAndMerge generator model. Many such references can exist in the same model hierarchy. Every such reference can be connected to different objects and different parameters. Furthermore, copies of the same SplitAndMerge generator model can exist in different models in different languages.

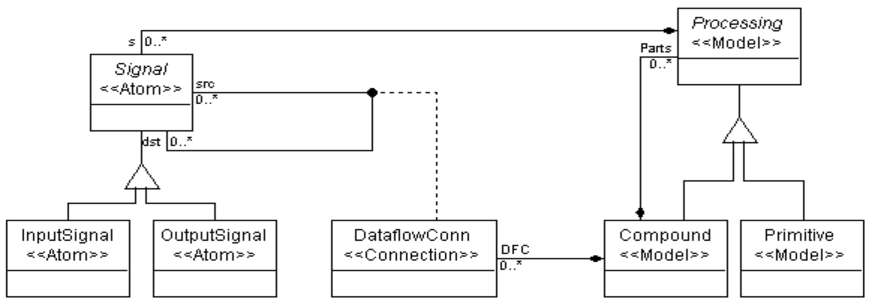

Figure 4. Hierarchical Signal Flow Metamodel

3. Metamodel Composition

Let us illustrate the process of adding generative modeling capability to an existing domain specific language through the image processing example. The metamodel of the hierarchical signal flow modeling language used in the example is shown in Figure 4.

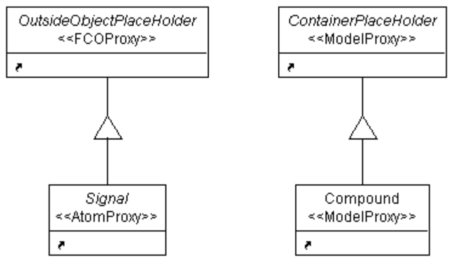

Figure 5. Metamodel Composition

Compounds are the composite models in this language; they can contain signal flow graphs themselves. Primitives are the leaf nodes in the hierarchy; they are the elementary signal processing components whose functionality is captured using a traditional programming language in a textual attribute. Both Compounds and Primitives have input and/or output ports called InputSignals and OutputSignals, respectively. They can be connected through DataflowConn connections. Processing models and Signal atoms are abstract components that help keeping the metamodel clear.

Figure 5 shows the metamodel that composes the signal flow language with generative modeling. The most frequently used technique in metamodel composition is inheritance. Here Signal inherits from OutsideObjectPlaceHolder, so that InputSignals and OutputSignals can be connected to StructuralPorts of Generator references. Compound inherits from ContainerPlaceHolder, so that Compounds can contain generative models.

In general, the user needs to decide which kind of objects should be able to participate in generative constructs and what models should contain these constructs. Then inheritance can be used to derive the generative capabilities from the placeholder objects. One way is to use simple inheritance as shown in Figure 5. Another option is to introduce a new concept, such as GenerativeCompound, and use multiple inheritance, for example to derive it from both ContainerPlaceHolder and Compound.

4. Generator Execution

The final representation issue concerns the generator script itself. There are many choices available for the language. For practical reasons, as GME uses Microsoft COM for component integration, we decided to use C++ with an API developed specifically for generative modeling in our prototype implementation.

The most convenient feature of this API is that the name of each structural port of the generator is automatically resolved to the object that is connected to the given port. Similarly, the name of each parameter port denotes an integer variable whose value is set to the value of the connected architectural parameter.

In fact, this generative API is just an extension of our high-level interpreter API for GME called the Builder Object Network (BON). While COM interfaces provide the interpreter writer all the functionality needed to access and manipulate the models, it entails using repetitious COM-specific querying, error checking and handling. To abstract these issues away from the interpreter writer, GME provides a collection of C++ wrapper classes: the Builder Object classes. When the user initiates model interpretation, the component interface builds a graph mirroring the models: for each model object an instance of the corresponding class is created. We refer to this graph as the Builder Object Network. The BON API provides all the necessary functionality to traverse the models along the containment hierarchy or any of the associations, to create and delete models and to get and set attributes, among others.

To simplify generator script writing, the generative API adds two groups of functions: one for duplicating model objects in a variety of ways and another for creating connections more easily. We present a sample script in Section 5.

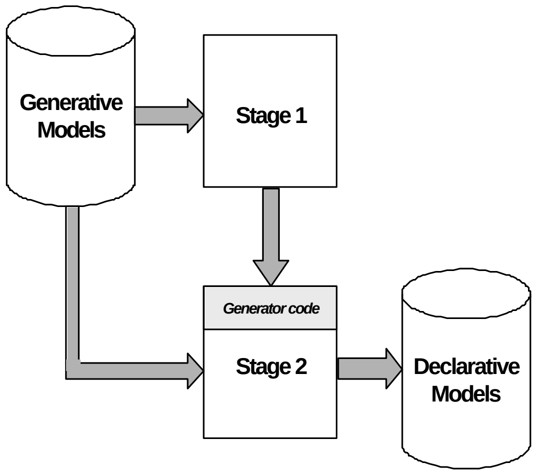

Figure 6. Generator Execution

Resolving generative models is a two-stage process as shown in Figure 6. First, code is generated for all generators and all generator references. This code becomes part of the second stage model interpreter that executes the generators and creates a new model hierarchy; one that has no generators, only pure declarative models. These models then can be used as any other domain model, i.e. all original model interpreters that were developed for the domain before generative capabilities were added are still fully supported.

In the first stage, a function is generated for every generator. The function body is the generator script as specified in the models. The argument list of the function mirrors the ports of the corresponding generator model; there is an argument for every structural port (of the generic FCO type) and one integer argument for every parameter port.

There is also a function generated for every generator reference, i.e. for every actual use of the generator. These functions are responsible for traversing the connections that are attached to the corresponding generator reference and obtaining the values of the architectural parameters. Then they simply call the generator function with the appropriate argument list.

These two sets of functions are compiled and linked together with the second stage model interpreter. This component traverses the models from the top down, creates the mirror image of all objects in a new blank root model and executes all generators, which, in turn, create new model objects. Note that generator execution is carried out in a bottom up fashion in the model containment hierarchy for efficiency reasons. This ensures that models created by the generators do not themselves contain generators that would need to be executed possibly multiple times.

5. Illustrative Example

Consider the data parallel image processing application introduced in Figure 1 and its generative representation shown in Figure 2. The generator script of SplitAndMergeGen is shown below:

```
#define MAXCHANNELS 32

if (Num < 1 || Num > MAXCHANNELS)
  return;

Atom  dst[MAXCHANNELS];
Atom  src[MAXCHANNELS];
Model proc[MAXCHANNELS];

std::string OutName= Out->getName();
std::string InName = In->getName();

dst[0] = Atom (Dst);
src[0] = Atom (Src);
proc[0]= Model (In->getParent());

for(int i = 1; i < Num; i++)
{
 src[i] =GAPI::portDup(Atom(Src),i);
 dst[i] =GAPI::portDup(Atom(Dst),i);
 proc[i]=GAPI::modelDup(owner,proc[0],i);
}

for(i = 0; i < Num; i++)
{
  GAPI::connect(owner,
    std::string("DFC"),
    src[i],
    proc[i], InName);

  GAPI::connect(owner,
    std::string("DFC"),
    proc[i], OutName,
    dst[i]);
}
```

Stage 1 of the interpreter creates the following function header for the script:

```
  void SplitAndMergeGen(
    Model owner,
    Object Dst,
    Object Src,
    Object Out,
    Object In,
    int Num
  )
```

and the following wrapper function for the generator reference SplitAndMerge shown in Figure 2:

```
void SplitAndMerge(Project pr)
{
 Object owner =
  Component::FindPort(pr, 6500000046);

 Object Dst =
  Component::FindPort(pr, 6600000093);

 Object Src =
  Component::FindPort(pr, 6600000092);

 Object Out =
  Component::FindPort(pr, 6600000095);

 Object In =
  Component::FindPort(pr, 6600000097);

 int Num = GenParameter (
   Component::FindPort(pr, 6600000085)
  ) -> GetValue();

 SplitAndMergeGen(owner,
   Dst, Src, Out, In, Num);
}
```

These functions become part of the Stage 2 interpreter. As it traverses the models, it eventually calls the SplitAndMerge function that finds all necessary objects that are connected to the generator reference, obtains the value for the Num parameter, and calls the SplitAndMergeGen function with the appropriate arguments.

The generator script above first verifies that Num falls within the acceptable range. Then it creates arrays of object pointers to refer to the output ports of SimpleSplitter, the input ports of SimpleMerger and to the Convolution models. Please refer to Figure 2. It initializes the first item of each array with the object that exists in the current model. Then it creates the necessary Num-1 many objects for each category. Finally, it makes the appropriate signal flow connections.

Conclusions

We presented an approach to generative modeling where architectural parameters and generator scripts are employed to specify model structure. The models are then used to automatically generate a declarative representation corresponding to a particular parameter instantiation. Generative modeling is particularly well suited to represent regular model structure and adaptive systems.

The primary design goal of our technique was reusability. This is accomplished by providing the generative modeling capability in a domain-independent fashion. The generative representation methodology is captured in a metamodel that can be easily incorporated into any domain-specific language through metamodel composition. The interpretation of the generative models is done through a two-stage domain-independent code generator/model interpreter.

Furthermore, the generator models and their scripts are also reusable within a model hierarchy or even across different domain-specific modeling languages, because the generators are self-contained modules that rely on generic model structure only and have no knowledge of and hence, dependence on any domain-specific concepts.

The ease of adding generative modeling capabilities to any modeling language was clearly demonstrated by the example. The only work that is needed is to specify where in the modeling language the generative capabilities are desired. This is a testimonial to the strength of Model Integrated Computing (MIC) in general, and the extensibility of the Generic Modeling Environment (GME) in particular.

References

[1] Sprinkle J., et al.: "The New Metamodeling Generation," IEEE Engineering of Computer Based Systems, Proceedings, Washington, D.C., USA, April, 2001.
[2] Ledeczi, A., et al.: "Composing Domain-Specific Design Environments," IEEE Computer, pp. 44–51, November 2001.
[3] UML Summary, ver. 1.0.1, Rational software corporation, et al., Sept. 1997
[4] Object Constraint Language Specification, ver. 1.1, Rational Software Corporation, et al., Sept. 1997.
[5] Ledeczi, A., et al.: "Model-Integrated Embedded Systems," in Robertson, Shrobe, Laddaga (eds) Self Adaptive Software, Springer-Verlag Lecture Notes in CS, #1936, February, 2001
[6] Ledeczi, A., et al.: "On Metamodel Composition," IEEE CCA 2001, CD-Rom, Mexico City, Mexico, September 5, 2001