Generating Aspect Code from Models

Jeff Gray[1], Ted Bapty[2], Sandeep Neema[2], Aniruddha Gokhale[2], Balachandran Natarajan[2]

gray@cis.uab.edu
{bapty, neemask, gokhale, bala}@isis-server.vuse.vanderbilt.edu

[1]University of Alabama at Birmingham
Computer and Information Sciences
Birmingham, AL
http://www.gray-area.org

[2]Vanderbilt University
Institute for Software Integrated Systems
Nashville, TN
http://www.isis.vanderbilt.edu

Abstract. Synthesis of source code from models typically proceeds as a direct mapping from each modeling element to the generation of a set of intentionally equivalent source code statements. When a library of components is available, the model interpreter can leverage a larger granularity of reuse by generating configurations of the available components. However, it is difficult to synthesize certain properties described in a model (e.g., those related to Quality of Service) due to the closed nature of the components, as available during generation-time. An aspect-oriented solution can provide the ability to instrument components with features that are specified in the model. This paper presents an idea, and example, for generating AspectJ code from domain-specific models.

1 Introduction

The traditional approach for generating artifacts from a domain-specific model involves the construction of an interpreter, or generator, which walks a tree-like representation of the model. The actions performed at each visited node result in the synthesis of a new representation of the model. Often, the generated artifact is represented as source code in a programming language. In such cases, the interpreter has built-in knowledge of the semantics of the domain and the programming language to which it is being mapped. The interpreter may also be aware of a library, or set of components, from which the synthesized implementation instantiates.

When an interpreter produces a source code artifact that relies on pre-existing libraries of components (i.e., the libraries are static and not a part of the generated artifact), it can be difficult to map properties of the model into the component. This is typically true even if the component library is available in source form (unless there is provision within the interpreter to parse and transform the component library itself during the model synthesis). The reason is that the component, during generation-time, is often treated as closed to modification – the granularity of the component representation is typically at the interface level, not the individual statements within the component implementation.

An aspect-oriented approach can assist in the generation of component customizations that extend the component with properties declared in the model. The focus of this position paper is to introduce a generation technique that relies on an aspect-oriented language to encode the extended features that are to be added to a component. In the next section, we introduce a modeling domain and an example that will motivate the problem. Section 3 presents an aspect-oriented solution, with the final section offering some concluding remarks.

2 Modeling BoldStroke Components in the GME

Boeing's BoldStroke is a product-line framework for avionics navigation software [Sharp, 1998]. There have been efforts within the DARPA MoBIES and PCES programs to model the structure, behavior, and interactions of subsets of applications built from BoldStroke components. A modeling effort for a subset of BoldStroke components has been conducted with the Generic Modeling Environment (GME) – a metaconfigurable CASE tool that supports the construction of domain-specific modeling environments [Lédeczi et al., 2001].

Figure 1 represents a simple model that contains five components. All of these components have specified parameters (e.g., frequency, latency, worst-case execution times) that affect Quality of Service (QoS) requirements. The first component is an inertial sensor. This sensor outputs the position and velocity deltas of an aircraft. A second component is a position integrator. It computes the absolute position of the aircraft given the deltas received from the sensor. It must at least match the sensor rate such that there is no data loss. The weapon release component uses the absolute position to determine the time at which a weapon is to be deployed. A mapping component is responsible for obtaining visual location information based on the absolute position. A map must be constructed such that the current absolute position is at the center of the map. A fifth component is responsible for displaying the map on an output device. The specific values of component properties will likely differ depending on the type of aircraft represented by the model (e.g., the latencies and WCETs for an F-18 would most likely be lower than a helicopter). The core modeling components describe a product family with the values for each property indicating the specific characteristics of a member of the family.

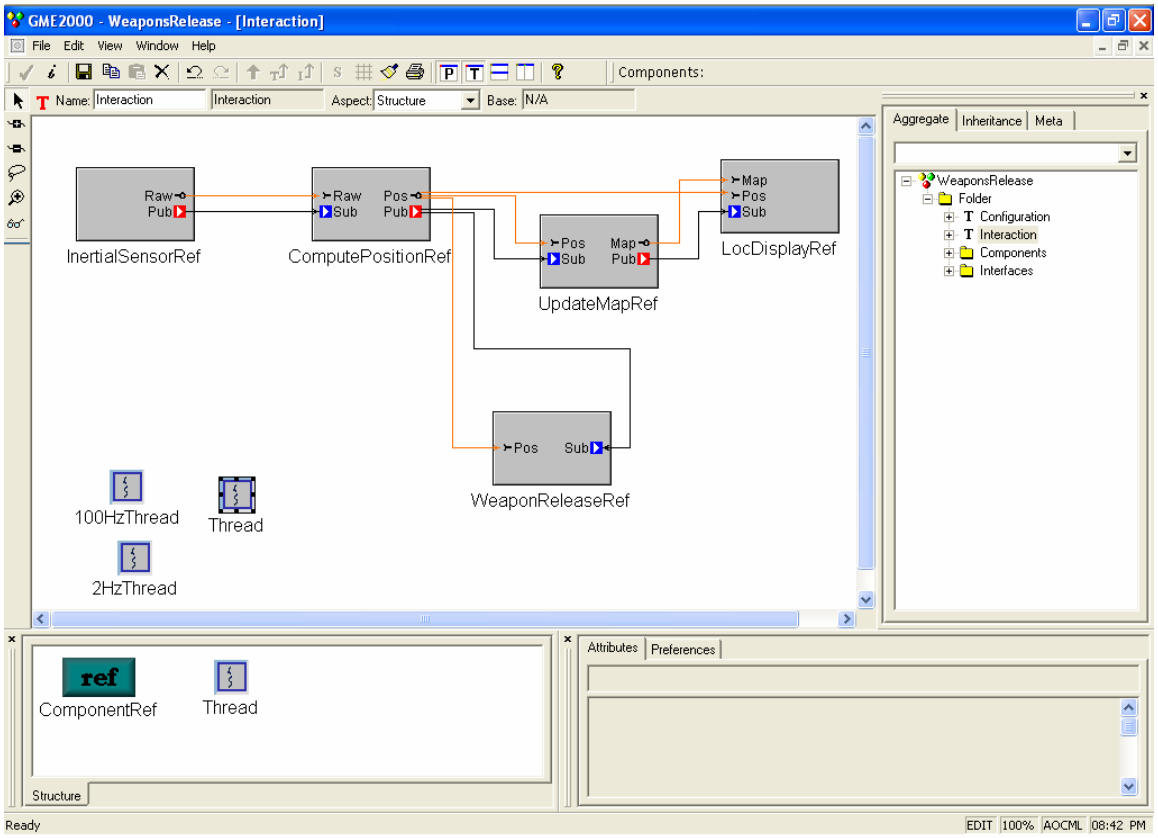

Figure 1: A GME Model of the Component Interactions

The internals of the components in Figure 1 permit the representation of CORBA Component Models (CCM) [Siegel, 2000]. The CCM provides capabilities that offer a greater level of reuse and flexibility for developers who need to deploy standardized components [Wang et al., 2001]. Each of the components in Figure 1 has internal details, in support of the CCM, that also are modeled. For instance, the contents of the Compute Position component are rendered in Figure 2.

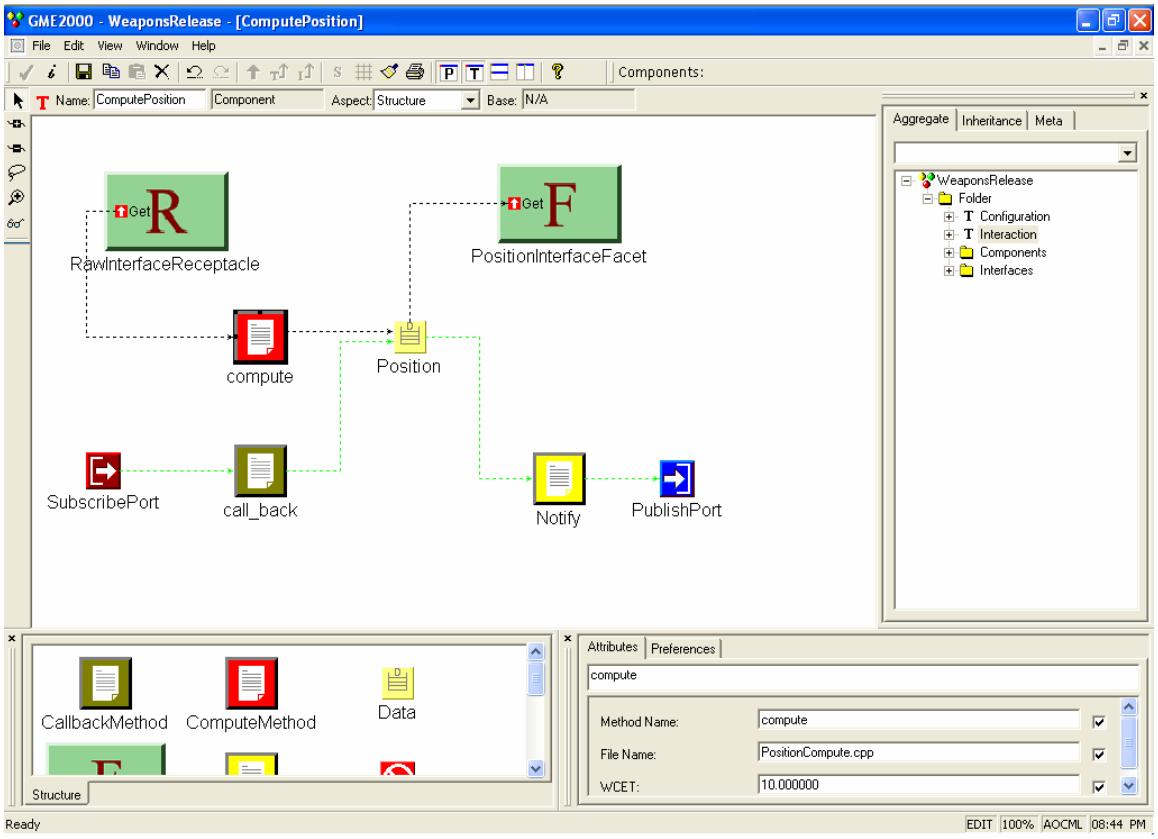

Figure 2: The Internals of Compute Position

Eager/Lazy Evaluation

In the interactions among the various components in the above example, there is a defined protocol for computing a value and notifying other components of a completed computation. These interactions are the result of a publish/subscribe model that uses an event channel. The typical scenario for these interactions is:

1. One component (C) receives an event from another component (S), indicating that a new value is available from S.
2. At some point in time, C invokes the `get_data` function of S in order to retrieve the most up to data value from S. C then performs a computation based upon the newly retrieved value.
3. At some point in time, component C notifies all of the other components that have subscribed to the event published by C.

There are situations where early acquisition and computation of data can waste resources. The determination concerning how often a computation should be made is an optimization decision. In an eager evaluation, all of the steps to perform the computation for a component are done at once. An eager evaluation would follow the three steps above in a strict sequential order each time an event is received from a supplier component. A lazy evaluation is less aggressive in computing the most up to date value. The second step, from above, is performed late. That is, the value from the supplier, and the actual computation, are only performed when a client component requests a data value. The computation is performed only when needed, not during each reception of an event from a supplier.

The determination of whether an eager/lazy strategy is to be used can be specified by properties in the model. The model interpreter can evaluate these properties during synthesis to generate the appropriate implementation constructs. As mentioned earlier, however, modifications to existing components may not be feasible during generation. The next section presents a brief example that demonstrates the use of an aspect-oriented language to implement a lazy strategy on one of the components in Figure 1.

3 Synthesizing Aspects

One of our goals is to demonstrate the capability for generating the configuration of BoldStroke components from domain-specific models in such a way that specific parts of each component are weaved together as an aspect. This goal fits well with the OMG's Model Driven Architecture (MDA) [Bézivin, 2001], [Burt et al., 2001] and also the idea of "fluid AOP" [Kiczales, 2001].

A possibility for realizing this objective would be the generation of AspectJ [Kiczales et al., 2001] code from models, as shown in Figure 3. In this figure, the model (top-left of figure) and specification aspects (top-right of figure) are sent through a weaver that constrains the model. Here, specification aspects represent the description of crosscutting concerns that are to be weaved into the model [Gray et al., 2001]. The constrained model (bottom-left of figure) can then be sent to a GME interpreter (bottom-right of figure) that generates the aspect code. This figure illustrates that there are two stages of weaving that are performed. A higher level of weaving is done on the model itself. This weaving instruments a base model with specific concerns (often represented as model constraints) that typically crosscut the model. The second type of weaving occurs from the aspect code that is synthesized and later processed by the AspectJ compiler. Thus, weaving at both the modeling level and the implementation level is achieved.

The amount of generated code produced from the aspect generator would actually be quite small. The assumption is that the core of the available components would already exist. Another assumption would be the existence of several different aspects of concern. These assumptions are in line with the work that other researchers are doing toward the goal of making a library of components and aspects available for a subset of the CORBA event-channel [Hunleth et al., 2001] in Java. As an alternative, the AspectC++ weaver could be used on the original C++ BoldStroke components [Mahrenholz et al., 2002]. For other languages, adaptations to a program transformation system (like the Design Maintenance System [DMS, 2002]) could be integrated within the model interpreter.

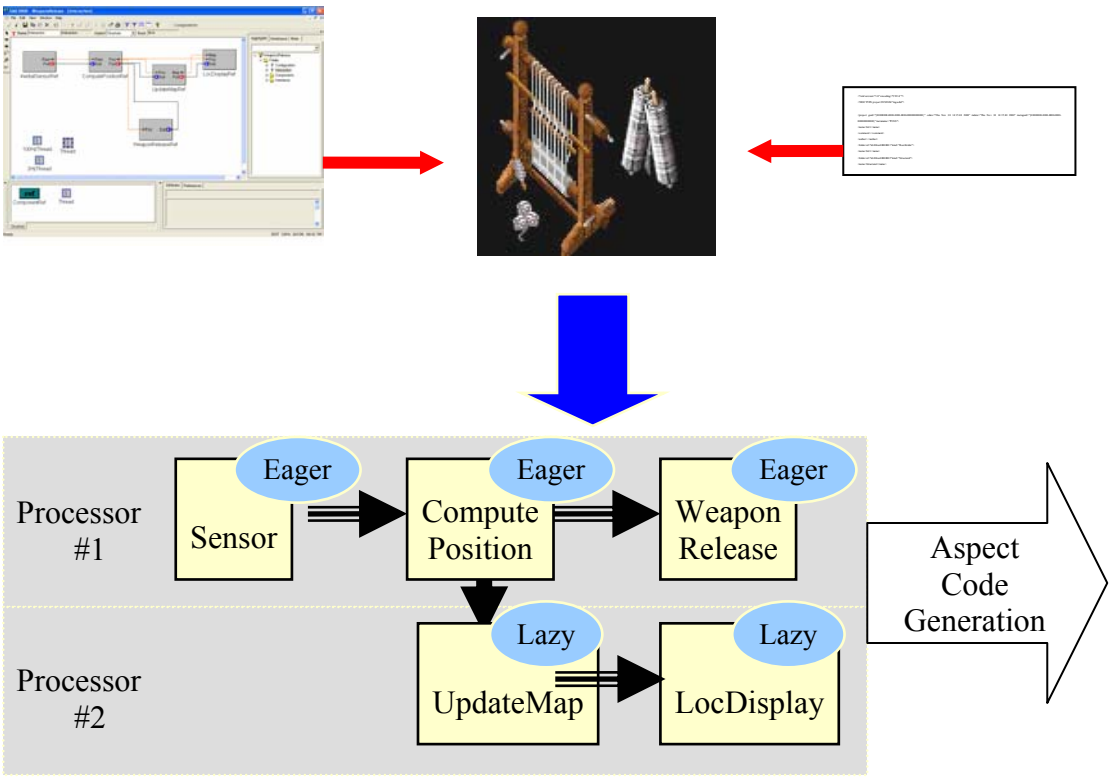

Figure 3: An MDA View of Aspect Code Generation

An example of a core library of components can be found in the Java code in Figure 4. This figure represents an abstract `Component` (a), and a `LocDisplay` component (b). The abstract component defines the required methods for the domain – the same methods that can be found in models like Figure 2. The `LocDisplay` subclass, for clarity, simply provides stubs for each of the method implementations.

```
public abstract class Component
{

  public abstract void call_back();
  public abstract int get_data();
  public abstract void init();

  public abstract void data_retrieve();
  public abstract void compute();
  public abstract void notify_availability();

  Protected int _data;

}
```

a) Component.java (from component library)

Figure 4: Base Class Java Components

```
public class LocDisplay extends Component
{

  public void call_back() {
    System.out.println("This was LocDisplay.call_back"); };

  public int get_data() { return _data; };

  public void init() { };

  public void data_retrieve() {
    System.out.println("This is LocDisplay.data_retrieve!");
    UpdateMap map = new UpdateMap();
    map.get_data();
  };

  public void compute() {
    System.out.println("This is LocDisplay.compute!"); };

  public void notify_availability() {
   System.out.println("This is LocDisplay.notify_availability!");

};
```

b) LocDisplay.java (from component library)
Figure 4 (cont.): Base Class Java Components

Example aspects are coded in Figure 5. The `Lazy` aspect contains abstract pointcuts. Other aspects (e.g., various other forms of Eager/Lazy, etc.) will refine the definition of the pointcuts through extension. It is assumed that the `Lazy` aspect would exist in a library of reusable aspectual components. This abstract aspect captures the model of lazy evaluation, as described earlier. The `call_back` "after" advice simply forwards all notifications on to client components without making any effort to retrieve data and compute the intention of the component.

The `LocDisplayLazy` aspect, from the figure below, manifests the type of code that is expected to be actually generated by the model interpreter. This code is very easy to generate. In fact, to synthesize the `LocDisplayLazy` aspect, all that is needed is the name of the class and the type of eager/lazy evaluation to weave. The code generator produces the concretized pointcuts that are needed to accomplish the weaving of the lazy evaluation concern with the `LocDisplay` component.

To summarize the idea, the assumption is that the code in Figures 4a, 4b, and 5a already exist. The synthesis step is to produce code, like that of Figure 5b, which represents the weaving of a particular concern as a result of some model property.

```
abstract aspect Lazy {

    abstract pointcut call_back(Component c);
    abstract pointcut get_data(Component c);

    after(Component c): call_back(c)
    {
      System.out.println("after:call_back (Lazy)!");
      c.notify_availability();
    }

    Before(Component c): get_data(c)
    {
      System.out.println("before:get_data (Lazy)!");
      c.data_retrieve();
      c.compute();
    }

}
```

a) Lazy Aspect (from aspect library)

```
aspect LocDisplayLazy extends Lazy {

    pointcut call_back(Component c) : this(c) &&
                        executions(void LocDisplay.call_back(..));

    pointcut get_data(Component c) : this(c) &&
                         Executions(int LocDisplay.get_data(..));

}
```

b) Concretization of Lazy Aspect with LocDisplay (generated)

Figure 5: Sample Strategies and Specification Aspects

4 Conclusion

Our position is that an aspect-oriented generative approach can be beneficial within the context of the MDA. We have already demonstrated in our previous work that AO techniques are beneficial when applied to the modeling process itself. Our future work will explore the possibilities for some of the ideas presented in this paper, with respect to the generation of AspectJ code to assist in the weaving of features into pre-existing components.

The assumptions of the ideas presented here are that the components are written in a language that has an associated aspect weaver. It is also necessary (for the approach to reach a mature status) to have an extensive library of reusable aspects. Much research is still needed, within the AO community in general, regarding the idea of large scale aspect reuse and composition.

Acknowledgement

We recognize the support of the DARPA PCES program in providing funding for the investigation of ideas presented in this paper.

References

[Bézivin, 2001] Jean Bézivin, "From Object Composition to Model Transformation with the MDA," *Technology of Object-Oriented Languages and Systems (TOOLS),* Santa Barbara, California, August 2001.

[Burt et al., 2002] Carol Burt, Barrett Bryant, Rajeev Raje, Andrew Olson, Mikhail Auguston, "Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models," *The 6th International Enterprise Distributed Object Computing Conference (EDOC)*, Switzerland, September 2002.

[DMS, 2002] http://www.semdesigns.com/

[Gray et al., 2001] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck, "Handling Crosscutting Constraints in Domain-Specific Modeling," *Communications of the ACM*, October 2001, pp. 87-93.

[Hunleth et al., 2001] Frank Hunleth, Ron Cytron, and Chris Gill, "Building Customized Middleware Using Aspect-Oriented Programming," *OOPSLA Workshop on Advanced Separation of Concerns*, Tampa, Florida, October 2001.

[Kiczales et al., 2001] Gregor Kiczales, Eric Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold, "Getting Started with AspectJ," *Communications of the ACM*, October 2001, pp. 59-65.

[Kiczales, 2001] Gregor Kiczales, "Aspect-Oriented Programming: The Fun Has Just Begun," *Software Design and Productivity Coordinating Group – Workshop on New Visions for Software Design and Productivity: Research and Applications*, Nashville, Tennessee, December 2001.

[Lédeczi et al., 2001] Akos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, November 2001, pp. 44-51.

[Mahrenholz et al., 2002] Daniel Mahrenholz, Olaf Spinczyk, Wolfgang Schröder-Preikschat, "Program Instrumentation for Debugging and Monitoring with AspectC++," *Proceedings of the The 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing* , Washington DC, USA, April/May 2002.

[Neema et al., 2002] Sandeep Neema, Ted Bapty, Jeff Gray, and Aniruddha Gokhale, "Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems," *First ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE '02)*, Pittsburgh, PA, October 6-8, 2002.

[Sharp, 1998] David Sharp, "Reducing Avionics Software Cost Through Component Based Product-Line Development," *Software Technology Conference*, Salt Lake City, Utah, April 1998.

[Siegel, 2000] Jon Siegel, *CORBA 3 Fundamentals and Programming*, John Wiley & Sons, 2000.

[Wang et al., 2000] Nanbor Wang, Doug Schmidt, and Carlos O'Ryan, "Overview of the CORBA Component Model," *Component-Based Software Engineering: Putting the Pieces Together*, George Heineman and William Councill, editors, Addison-Wesley, 2001.