Appeared in: Aspect-Oriented Software Development, pp. 681-705, Addison, Wesley, 2004

TWO-LEVEL ASPECT WEAVING TO SUPPORT EVOLUTION IN MODEL-DRIVEN SOFTWARE

JEFF GRAY
JANOS SZTIPANOVITS
DOUGLAS C. SCHMIDT
TED BAPTY
SANDEEP NEEMA
ANIRUDDHA GOKHALE

> *An important step in solving a problem is to choose the notation. It should be done carefully. The time we spend now on choosing the notation may be well repaid by the time we save later avoiding hesitation and confusion. Moreover, choosing the notation carefully, we have to think sharply of the elements of the problem which must be denoted. Thus, choosing a suitable notation may contribute essentially to understanding the problem.* (Polya, 1957)

Since the inception of the software industry, models have been a beneficial tool for managing complexity. In fact, the first commercial software package that was sold independent of a hardware manufacturer was an application for constructing flow chart models, i.e., ADR's AUTOFLOW (ADR, 2002). In numerous disciplines, models are constructed to assist in the understanding of the essential characteristics of some instance from a particular domain. Mechanical engineers, architects, computer scientists, and many other professionals create models to provide projected views over an entity that has been abstracted from the real-world. As tools for creative exploration, even children erect models of real-world structures using Legos, Tinker Toys, and other similar materials.

As Polya points out in the opening quote, the notation chosen to represent our abstractions contributes to the ease, or difficulty, with which we understand the essence of a problem. Selecting the correct modeling abstractions can make the difference between a helpful aid or a hindrance to comprehension. In models for computer-based systems, tool support can also offer assistance in comprehending complex systems.

In addition to improving comprehension, models are also built to explore various design alternatives. In many domains, it is often too costly (in both time and money) to build variations of the real product in order to explore the consequences and properties of numerous configuration scenarios. For example, a model of the Joint Strike Fighter aircraft, along with configurations of various hostile scenarios, permits the simulation of an aircraft before it has even left the production line (Vocale, 2000). Models can be the source for simulations or analyses that provide a more economical means for observing the outcome of modified system configurations. The level of maturity of a chosen modeling tool can greatly influence the benefits of the

modeling process, which is especially true when a modeling tool can make changes throughout a model's lifecycle.

As Gerald Sussman observes (Sussman, 1999), in traditional system development, "Small changes in requirements entail large changes in the structure and configuration." A desirable characteristic is to have a change in the requirements be proportional to the changes needed in the corresponding implementation. Unfortunately, crosscutting requirements (such as high availability, security, and scalability in distributed systems) tend to have a global impact on system structure, which is hard to manage.

Our work involves the construction of models that represent a system in a particular domain. From these domain-specific models of systems and software, various artifacts are generated, e.g., source code, or even simulation scripts. We have found that model-based approaches can help to solve problems that often accompany changes to system requirements. For example, (Neema et al., 2002) offer an approach for synthesizing models represented as finite state machines into a contract description language, which is then translated to C++. The benefit of this technique is that very small

changes to the state machine models often result in large transformations of the representative source code. Thus, a single manipulation of a higher-level abstraction may correspond to multiple transformations at the concrete level, resulting in a conservation of effort when compared to the equivalent effort needed to make the same modification at the implementation level.

As evidenced by the topics covered in other chapters of this book, the idea of Aspect-Oriented Software Development (AOSD) is growing in depth and breadth. The techniques espoused by AOSD researchers generally provide new capabilities for modularizing crosscutting concerns that are hard to separate using traditional mechanisms. This chapter summarizes our work in applying AOSD techniques to domain-specific modeling and program synthesis. The use of *weavers*, which are translators that perform the integration of separated crosscutting concerns, is described at multiple levels of abstraction. Our research on Aspect-Oriented Domain Modeling (AODM) employs the following two-level approach to weaving:

- At the top-level, weavers are built for domain-specific modeling environments. The concept of applying AOSD techniques to higher lev-

els of abstraction is covered in Section 1, where we describe our Constraint-Specification Aspect Weaver (C-SAW)[1]. This section also provides an overview of Model-Integrated Computing (MIC). An example of AODM is described in Section 2.

- The second level of weaving occurs during model interpretation. Synthesis of source code from models typically proceeds as a mapping from each modeling element to the generation of a set of semantically 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. It is hard, however, to synthesize certain properties described in a model, e.g., those related to quality of service (QoS), due to the closed nature of the components. An aspect-oriented solution can provide the ability to instrument components with features that are specified in the model. Section 3 presents an approach and an exam-

[1] According to *Webster's Revised Unabridged Dictionary*, a crosscut saw (or c-saw,) is "A saw, the teeth of which are so set as to adapt it for sawing wood crosswise of the grain rather than lengthwise."

ple for generating AspectJ (Kiczales et al., 2001) source code from domain-specific models.

The chapter concludes with summary remarks and a description of current and future work in this area.

X.1 MODEL-INTEGRATED COMPUTING AND AOSD

> *To support this focus on the development of interacting subsystems with multiply-redundant design requires the development of languages that allow description of the function and relationships between different parts of the overall system. These descriptions "let go" of the specific logic of individual processes to capture the interactions that are necessary for the redundancy and robustness of multiple processes. When stated in this way we see that it is the description of constraints between functional units of the system that are the essential parts of the collective description of the system.* (Sussman, 1999)

The aim of Domain-Specific Modeling (DSM) is similar to the objective of textual domain-specific languages (DSL) (van Deursen et al., 2000) in that expressive power is gained from notations and abstractions aligned to a specific problem domain. A DSM approach typically employs graphical representations of the domain abstractions rather than the textual form of a tradi-

tional DSL. A program in a DSL is also usually given a fixed interpretation, but a model created from DSM may have multiple interpretations. For example, one interpretation may synthesize to C++, whereas a different interpretation may synthesize to a simulation engine or analysis tool.

Like DSLs, domain-specific modeling raises the level of abstraction to highlight the key concerns of the domain in a manner that is intuitive to a subject matter expert or systems engineer. A Domain-Specific Visual Language (DSVL) (DSVL03, 2003) can decouple designers from specific notations, such as UML (Booch et al., 1998). In domain-specific modeling using a DSVL, a design engineer describes a system by constructing a visual model using the terminology and concepts from a specific domain.

X.1.1 Model-Integrated Computing

An approach called Model-Integrated Computing (MIC) (Karsai, 1995) has been refined at Vanderbilt University over the past decade to assist the creation and synthesis of computer-based systems. A key application area for MIC is those domains (such as embedded systems areas typified by automo-

tive and avionics systems) that tightly integrate the computation structure of a system and its physical configuration. In such systems, MIC has been shown to be a powerful tool for providing adaptability in frequently changing environments. An example of the flexibility provided by MIC is documented in (Long et al., 1998), where an installed system at the Saturn automobile factory was shown to offer significant improvements in throughput by being able to adapt to changes in business needs and the physical environment. Other example domains where MIC has been successfully applied are the DuPont chemical factory (Garrett et al., 2000), numerous government projects supported by DARPA and NSF, electrical utilities (Moore et al., 2000), and even courseware authoring support for educators (Howard, 2002).

A specific instance of the type of domain-specific modeling supported by MIC is implemented using the Generic Modeling Environment (GME) (Lédeczi et al., 2001). The GME is a modeling environment that can be configured and adapted from meta-level specifications (called the *modeling paradigm*) that describe the domain (Nordstrom et al., 1999). When using

the GME, a modeler loads a modeling paradigm into the tool to define an environment containing all the modeling elements and valid relationships that can be constructed in a specific domain.

The process for applying MIC is shown in Figure 1. The left-hand side of this figure describes the task of creating new modeling environments. From meta-level specifications, new modeling environments are generated using meta-level translators (note that this process is self-descriptive – the meta-level specifications are also created with the GME). These meta-level specifications define the domain *ontology*, the specifications that identify the pertinent entities of the domain, as well as their related associations.

After a modeling environment has been generated, a domain expert can then create models for the particular domain associated with the environment (see the middle of Figure 1). Once a model has been created, it can then be processed by domain interpreters, which traverse the internal data structures that represent the model and generate new artifacts. These interpreters can synthesize an application that is customized for a specific execution plat-

form, as well as generate input to analysis tools. The synthesis/interpretation task is represented by the right-hand side of Figure 1.

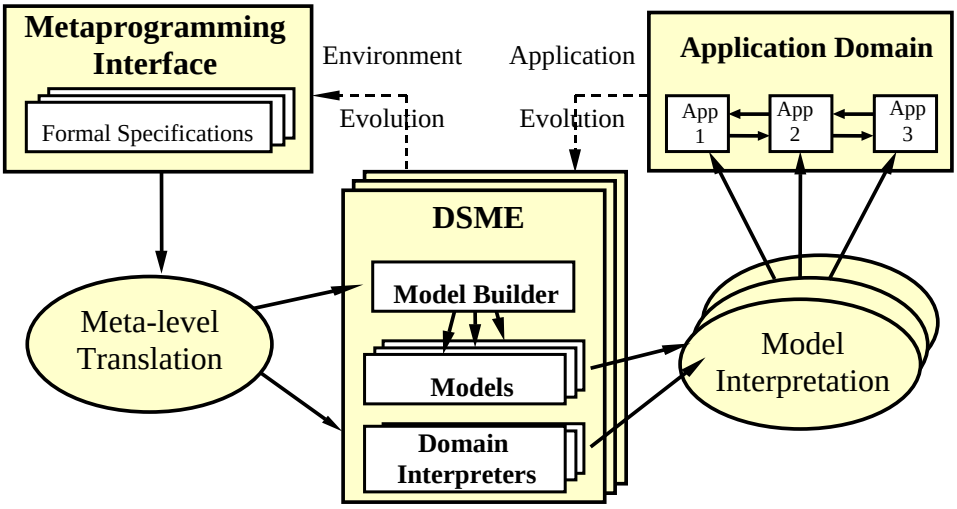


Figure 1: Process for Applying Model-Integrated Computing

An example of a meta-model is shown in Figure 2. In the GME, a meta-model is described with UML class diagrams and constraints that are specified in the Object Constraint Language (OCL) (Warmer and Kleppe, 1999). At the meta-modeling level, OCL constraints are used to specify the semantics of the domain that cannot be captured with the static relationships defined by a class diagram. The meta-model of Figure 2 specifies the entities

and relationships among collaborators in a middleware publisher/subscriber service, such as a CORBA event channel (Harrison et al, 1997). For instance, the meta-model contains the representation of several types of connecting ports, as well as various methods (e.g., call-back or notify) that are needed to realize the event channel. Constraints are not explicitly shown in this screenshot, but an informal example of a constraint for Figure 2 would state, "Every data object that is attached to a call-back and compute method must also be associated with a corresponding notify method." The meta-model can itself be interpreted to produce a new modeling environment. In fact, this particular meta-model defines the ontology for the subset of the Bold Stroke avionics models that we present in Section 2, i.e., Figure 6 is an instance of the meta-model of Figure 2. The environment generated from this meta-model will provide semantic checks to ensure that the constructed models conform to the semantics of the meta-model (Sztipanovits et al., 2002). Other mature meta-modeling environments include MetaEdit+ (Pohjonen and Kelly, 2002) and DOME (DOME, 2003). A similar approach that also

also uses OCL has been adopted recently in the Kent Modeling Framework (KMF) (KMF, 2003).

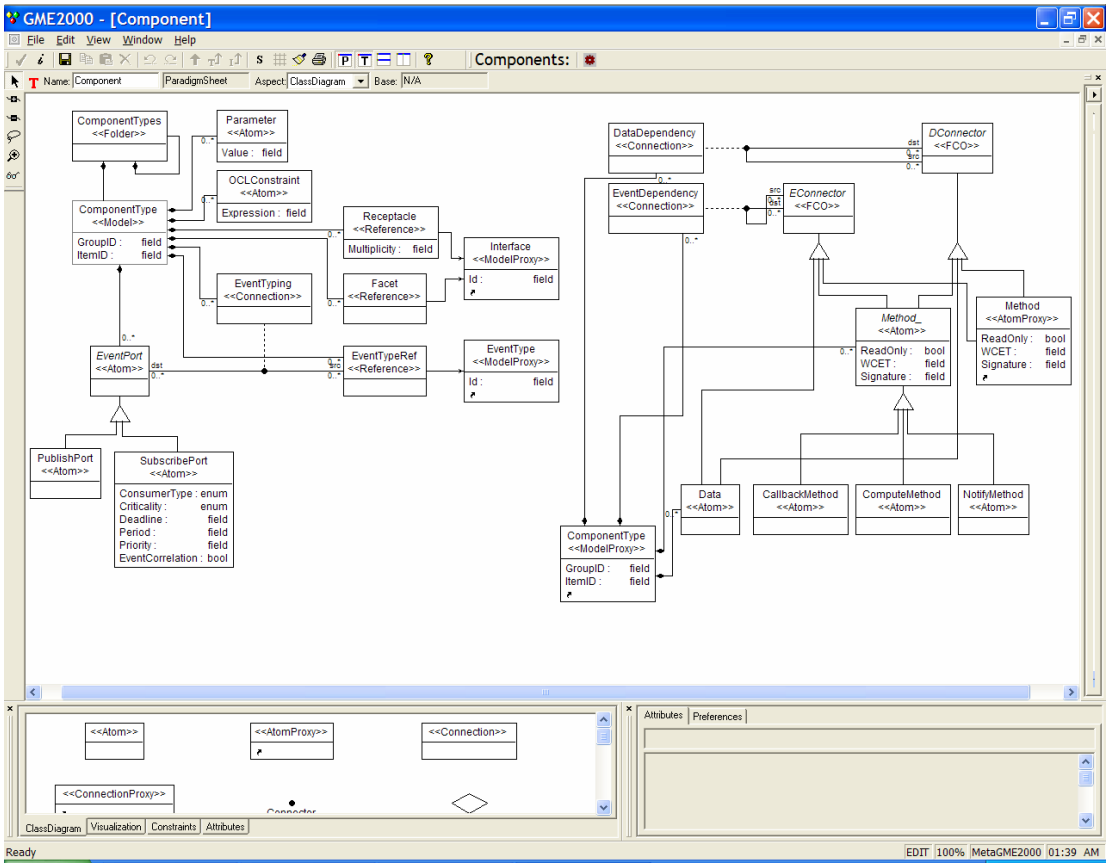


Figure 2: A GME Meta-model for Bold Stroke Avionics Mission Computing

X.1.2 Crosscutting Concerns in Domain Modeling

As described in other chapters of this book, a distinguishing feature of AOSD is the notion of crosscutting, which characterizes the phenomenon whereby some representation of a concern is scattered among multiple boundaries of modularity, and tangled amongst numerous other concerns. Aspect-Oriented Programming (AOP) languages, such as AspectJ (Kiczales et al., 2001), permit the separation of crosscutting concerns into aspects. We have found that the same crosscutting problems that arise in code also exist in domain-specific models (Gray et al., 2001). For example, it is often the case that the meta-model forces a specific type of decomposition, such that the same concern is repeatedly applied in many places, usually with slight variations at different nodes in the model (this is a consequence of the "dominant decomposition" (Tarr et al., 1999), which occurs when a primary modularization strategy is selected that subjects other concerns to be described in a non-localized manner).

A concrete example of crosscutting in models will be shown in Section 2 based on the meta-model in Figure 2. An abstract illustration of the effect

of crosscutting constraints is presented in Figure 3. In this figure, a hierarchical decomposition is evident. Yet, such a decomposition has forced another concern (represented by the checkered boxes, all pointing toward the existence of a global crosscutting constraint) to be distributed across the hierarchy of the model. This results in much redundancy and replicated structure because the concern is tailored to the context of numerous nodes in the model.

There are several different types of constraints that may be applied throughout a model. Figure 4 shows a set of resource constraints that indicate specific hardware resources needed by software. Several of the models created using the GME tool contain thousands of components, with several layers of hierarchy. In the presence of a dominant decomposition, the constraints of a complex model become tangled throughout the model, which makes them hard to understand. The AODM approach can isolate the crosscutting constraints to modularize these global system properties more effectively.

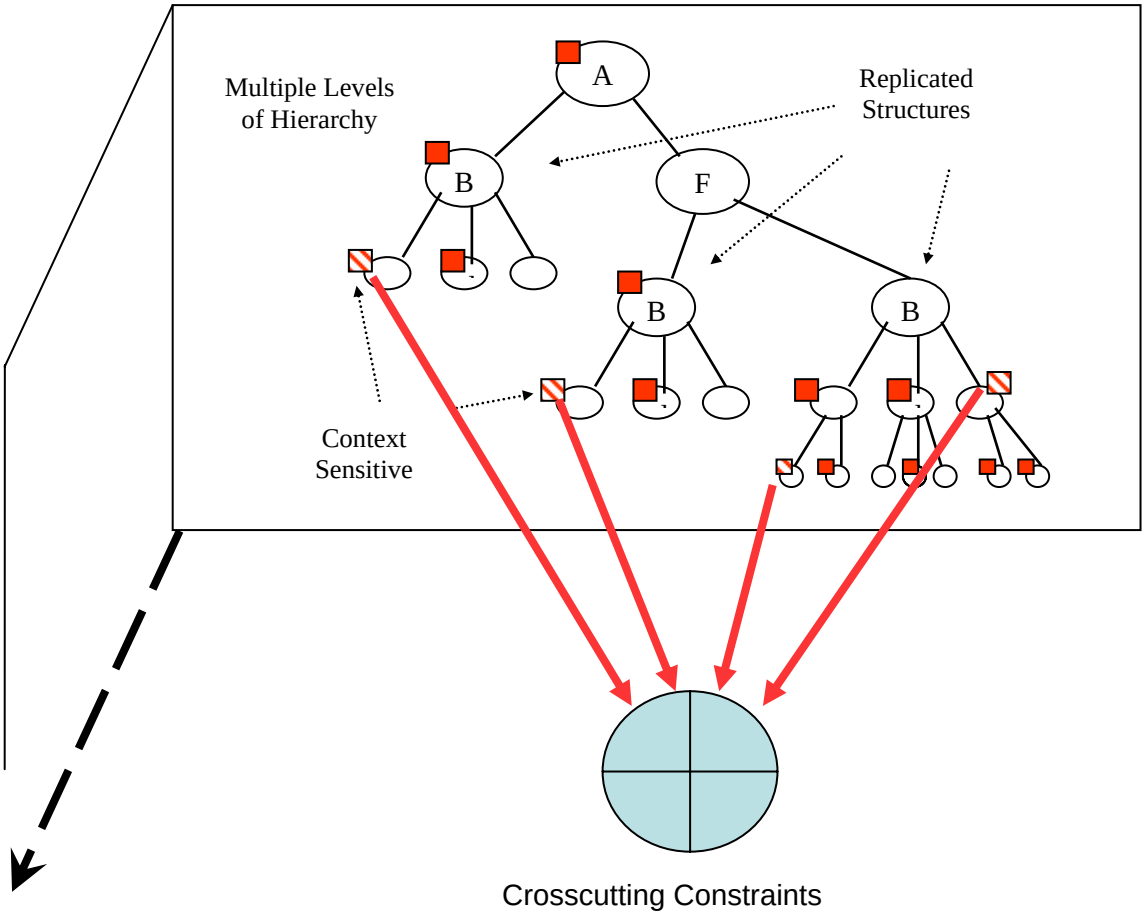
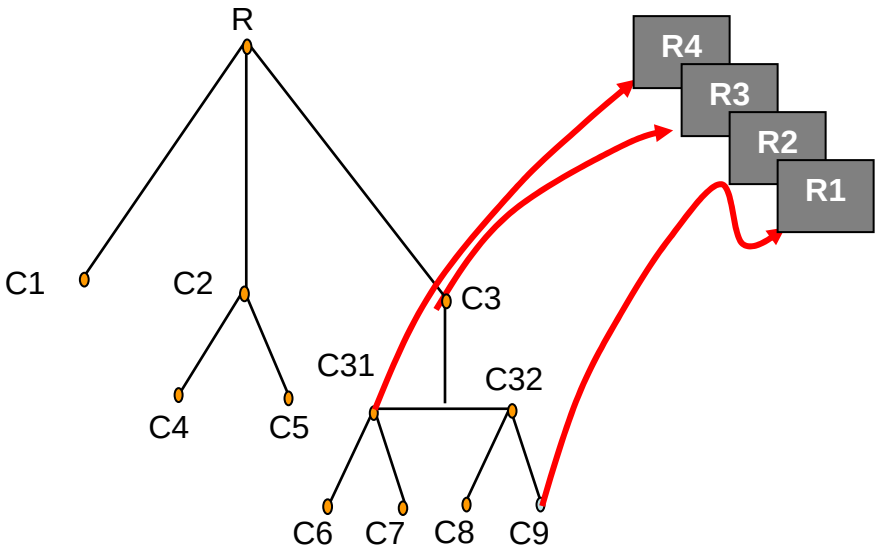


Figure 3: Crosscutting Constraints in Domain-Specific Modeling




```
Constraint Resource () {
      C31.assignedTo(resources()->R4) implies
      (C3.assignedTo(resources()->R3) and
        ((C9.assignedTo(resources()->R1) or
         (C9.assignedTo(resources()->R1))}
```

Figure 4: Crosscutting Resource Constraints

X.1.3 Model Weavers

Our approach to AODM requires a different type of weaver from those that others have constructed in the past, e.g., the AspectJ weaver (Kiczales et al., 2001), because the type of software artifact processed by our model weaver

differs from traditional programming language weavers. Programming language weavers support better modularization at a lower level of abstraction by processing source code, but a domain-specific modeling weaver processes the structured description of a visual model. In particular, this new weaver requires the capability of reading a model that has been stored in the Extensible Markup Language (XML). This weaver also requires the features of an enhanced constraint language. The standard OCL is strictly a declarative language for specifying assertions and properties on UML models. Our need to extend the OCL is motivated by the fact that we require an imperative language for describing the actual model transformations. We have created the Constraint Specification Aspect Weaver (C-SAW) to provide support for modularizing crosscutting modeling concerns in the GME.

Our approach to model weaving involves the following concepts:

Model Constraints: This type of constraint appears as attributes of modeling elements. It is these constraints that are traditionally scattered across the model. These constraints are typically represented by a specialized entity in the meta-model, e.g., the `OCLConstraint` meta-type in Figure 2.

Modeling Pointcuts: A *modeling pointcut* is a new modular construct that specifies crosscutting concerns across a model hierarchy. Each modeling pointcut describes the binding and parameterization of strategies to specific nodes in a model. A modeling pointcut is conceptually similar to a pointcut in AspectJ (Kiczales et al., 2001). Like an AspectJ pointcut designator, a modeling pointcut is responsible for identifying the specific locations of a crosscutting concern and offers the capability to make quantifiable statements across the boundaries of a model (Filman and Friedman, 2000). Quantification permits statements such as, "For all the places where properties X, Y, and Z hold, then also make sure that property A and B are also true, but not property C." In the context of modeling pointcuts, the general notion of quantification refers to the ability to make projected assertions and transformations across a space of conceptual representation, e.g., models or even source code.

Strategies: A *strategy* is used to specify elements of computation, constraint propagation, and the application of specific properties to the model nodes (Note: we refer to "model nodes" as being those modeling elements

that have a definition in the meta-model and serve as visualization elements in the domain model). The name “strategy” is inspired by the strategy design pattern (Gamma et al., 1995). We use this term to define a collection of interchangeable heuristics. Strategies are generic in the sense that their descriptions are not bound to particular model nodes. Each weaver that supports a specific meta-level GME paradigm will have disparate strategies. A strategy provides a hook that the weaver can call to process node-specific constraint application and propagation. Strategies therefore offer numerous ways for instrumenting nodes in the model with crosscutting concerns. Section 2.2 of this chapter provides an example strategy for assigning eager/lazy evaluation within a CORBA event channel.

The three items listed above differ in purpose and in application, yet each is based on the same underlying language. We call this language the Embedded Constraint Language (ECL). The ECL is an extension of the OCL and provides many of the common features of OCL, such as arithmetic operators, logical operators, and numerous operators on collections (e.g., `size`, `forAll`, `exists`, `select`). A unique feature of ECL that is not

provided within OCL, however, is a set of reflective operators for navigating the hierarchical structure of a model. These operators can be applied to first class model objects (e.g., a container model or primitive model element) to obtain reflective information needed in either a strategy or pointcut.

Figure 5 shows the use of model weaving with C-SAW. In this figure, the solid arrows represent the output from tools that generate, or transform, a model. The GME can export the contents of a model in the form of an XML document (see step "1" in Figure 5; in this case, the exported XML is related to the meta-level paradigm from which the model was constructed, such as the one in Figure 2). In our approach, the exported XML representing a model is often devoid of any constraints. The constraints are not present in such cases because they are modularized by the pointcuts and strategies.

The input to the domain-specific weaver consists of the XML representation of the model, as well as a set of modeling pointcuts provided by the modeler (step "2"). In Figure 5, these entities are positioned to the left of the domain-specific weaver. The output of the weaving process is a new XML description of the model (step "3"). This enhanced model, though, contains

new concerns that have been integrated throughout the model by the weaver, and can be reloaded into the GME (step "4").

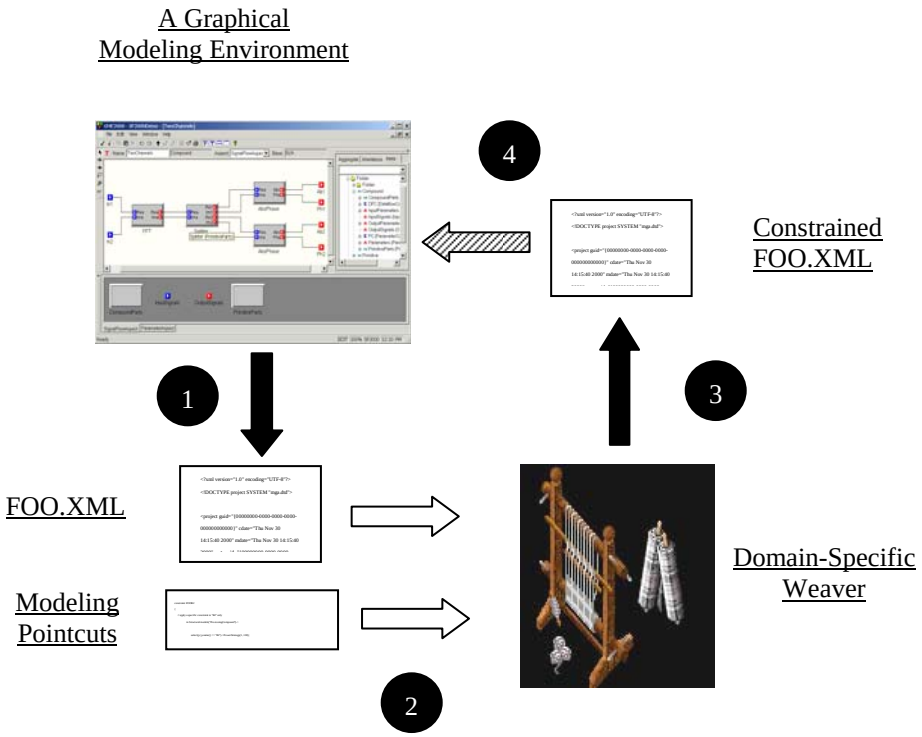


Figure 5: Process of Using a C-SAW Model Weaver with the GME

There are several key benefits of the aspect-oriented (AO) approach described above. For example, consider the case of modeling an embedded system where constraints often have conflicting goals (e.g., latency and resource usage). In a non-AO approach, latency and resource requirements would be

scattered and tangled throughout the model. As a result, it would be hard to isolate the effects of latency or resource constraints on the design. By using aspects to represent these concerns, however, designers can apply modeling pointcuts separately to see how the system is affected in each case. In this way, areas of the system that will have more difficulty meeting a requirement may be given more relaxed constraints, and other parts of the system may be given tighter constraints.

In general, AODM enables designers to isolate and study the effects of concerns (such as constraints) across an entire model. This approach is desirable with respect to application-constraint tuning, i.e., the separation of concerns provided by the modeling pointcuts improves the modular understanding of the effect of each constraint. The plugging/unplugging of various sets of modeling pointcuts into the model can be described as creating "what if" scenarios. These scenarios help explore constraints that may have conflicting goals. The insertion and removal of design alternatives is analogous to AspectJ's ability to plug/unplug certain aspects into a core piece of Java code (Kiczales et al., 2001).

Our previous work (Gray et al., 2001) investigated the idea of a meta-weaver framework that used generative programming techniques (Czarnecki and Eisenecker, 2000) to produce new model weavers based upon the strategies specified for a specific domain. The framework could therefore be instantiated to produce a specific weaver for a particular domain (e.g., avionics) and could also be instantiated with different strategies to generate another weaver for an additional domain (e.g, automotive electronics). The details of the meta-weaver framework are described in (Gray et al., 2001).

X.2 Example: Model Weaving of Eager-Lazy Evaluation Constraints

> *The point of time at which the resources are acquired can be configured using different strategies. The strategies should take into account different factors, such as when the resources will be actually used, the number of resources, their dependencies, and how long it takes to acquire the resources. Regardless of what strategy is used, the goal is to ensure that the resources are acquired and available before they are actually used.* (Kircher, 2002)

This section introduces a modeling domain and an example to explain the process of model weaving with C-SAW. Section 3 expands on this example by presenting an approach for generating AspectJ code from models.

X.2.1 Modeling Bold Stroke Components in GME

Boeing's Bold Stroke project (Sharp, 1998) uses COTS hardware and middleware to produce non-proprietary, standards-based component architecture for avionics mission computing capabilities, such as heads-up display, navigation, data link management, and weapons control. A driving objective of Bold Stroke was to support reusable product-line applications (Clements and Northrop, 2001), leading to a highly configurable application component model and supporting middleware services. There have been efforts within the DARPA MoBIES and PCES programs to model the structure, behavior, and interactions of subsets of applications built from Bold Stroke components. A modeling effort for a subset of Bold Stroke components has been conducted using the GME.

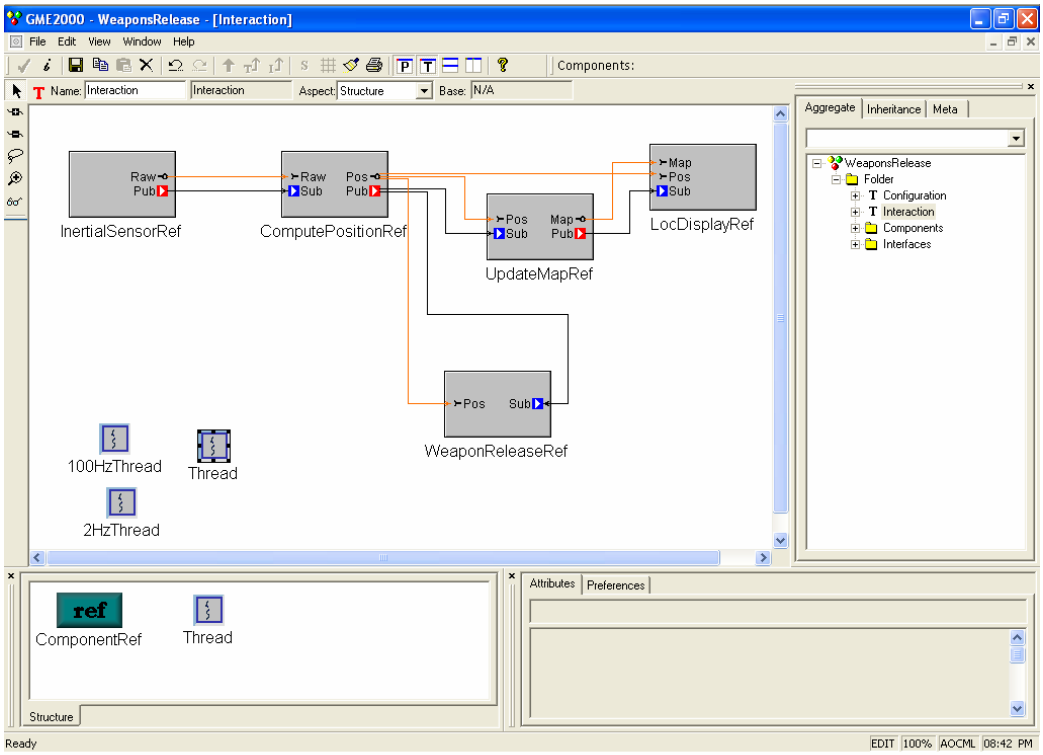


Figure 6: A GME Model of the Bold Stroke Component Interactions

Figure 6 represents a simple model that contains five components. All of these components have specified parameters (e.g., frequency, latency, Worst-Case Execution Time (WCET)) that affect end-to-end 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 at which time to deploy a weapon. 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 are often lower than those of 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. A more detailed description of WCET within the context of Bold Stroke can be found in (Gu and Shin, 2003).

The internals of the components in Figure 6 permit their realization using the CORBA Component Model (CCM) (Siegel, 2000). The CCM provides capabilities that offer a greater level of reuse and flexibility for developers to deploy standardized components (Wang et al., 2001). Each of the

components in Figure 6 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 7. This figure specifies the interactions of entities within a middleware event channel, e.g., call-back function, notification procedure, local data store (Position). The ports and Receptacle/Facet entities provide the connection points to other components and events.

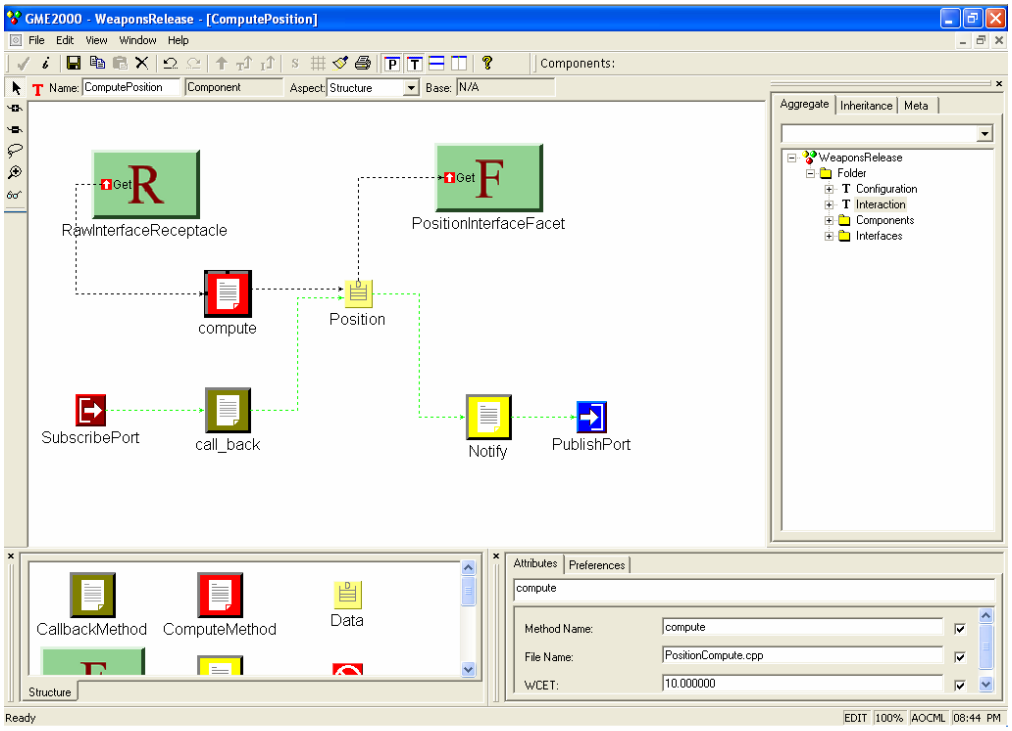


Figure 7: The Internals of the Compute Position Component

X.2.1.1 Eager/Lazy Evaluation

In the interactions among the various components in the weapons deployment example, there is a 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 the CORBA Event Service, which consists of suppliers who publish events to an event channel, which then delivers the events to the appropriate consumers in a timely and reliable manner (Harrison et al., 1997). 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. C then invokes the `get_data()` function of S to retrieve the most recent data value from S. C then performs a computation based upon the newly retrieved value.
3. Component C subsequently notifies all of the other components that subscribed to the event published by C.

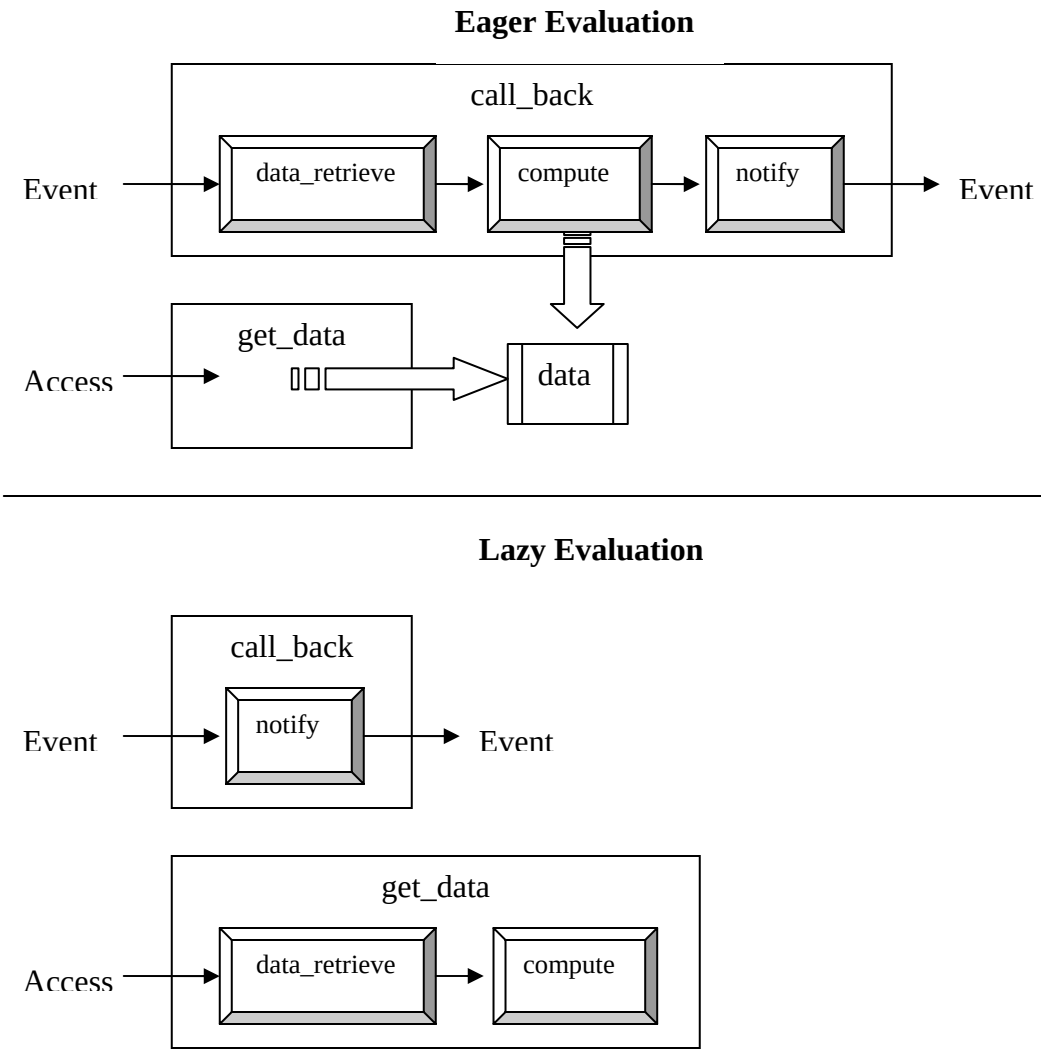


Figure 8: Description of Eager/Lazy Strategy

Because 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, as described below:

- In an eager evaluation, all 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 (see the top part of Figure 8 for a depiction of the eager evaluation protocol) each time an event is received from a supplier component.
- A lazy evaluation is less aggressive in computing the most recent value. The second step, from above, is performed late, i.e., the value from the supplier and the actual computation, are performed *after* a client component requests a data value. The computation is therefore performed only when needed, not during each reception of an event from a supplier. The concept of a lazy evaluation is shown in the bottom part of Figure 8.

X.2.2 Strategies for Eager/Lazy Evaluation

The manner by which a determination of eager/lazy evaluation is made can be modeled as an aspect. The determination is typically made according to some optimization protocol, which is spread across each component of the model. If it is essential to change properties of the model, it would be necessary to revisit each modeling element and modify the eager/lazy assignment of each node. The dependent nature of the eager/lazy evaluation on properties in the model makes change maintenance a daunting task for non-trivial models.

It would be useful to be able to separate the criteria used to assign an evaluation. Such separation would support changeability and exploration of different protocols. A specific strategy for determining eager/lazy evaluation is given in Figure 9. This figure shows how the `EagerLazy` strategy simply determines the location of the start and end nodes of a range of elements within the model to which the strategy is applied. It also finds the context of folders and models that will be needed during the distribution of the concern. The parameterization of the start and end nodes – and also the latency

threshold – enables this strategy to be called by a modeling pointcut in numerous ways. This AO design permits the weaving of different constraints into the model in a more efficacious manner by quantifying over the model space and parameterizing the heuristic that is applied. Without such capabilities, a modeler would have to visit every node of the model that is affected by the concern and manually apply the modification.

The `DetermineLaziness` strategy is invoked on the start node (because the strategy works backwards, the start node is actually the node that is nearest to the end of the interaction). This strategy performs a simple computation to determine the evaluation assignment for the current node. The intent of the strategy is to assign a component to an eager evaluation until the latency threshold is exceeded. After the threshold is exceeded, all subsequent components are assigned as lazy. If the current node is not the end node of the interaction, then the strategy named `BackFlow` is fired (this strategy is not shown in order to conserve space). The `BackFlow` strategy collects all of the suppliers of the current node (this is done by finding the components

that are on the current component's data flow, and serve as suppliers) and invokes a continuation on the collection.

```
defines EagerLazy, DetermineLaziness;

strategy EagerLazy(EndName : string; latencyThreshold : integer)
{

  declare components, interactions, startNode, endNode : node;

  components := findFolder("Components");
  interactions := findModel("Interaction");

  startNode := self;
  endNode := components.findModel(EndName);

  startNode.DetermineLaziness(components, interactions, endNode,
                              latencyThreshold);

}

strategy DetermineLaziness(components, interactions, endNode : node;
                           latencyThreshold : integer)
{

  declare static accumulateLatency : integer;
  declare latency : integer;
  declare currentID, endID : string;

  if (accumulateLatency < latencyThreshold) then
   AddConstraint("EagerLazy", "assignment = lazy");
  else
   AddConstraint("EagerLazy", "assignment = eager");
  endif;

  latency := self.compute.latency;
  accumulateLatency := accumulateLatency + latency;

  getID(currentID);
  endNode.getID(endID);

  if(currentID <> endID) then

     self.BackFlow(components, interactions, endNode,
                   latencyThreshold);

  endif;

}
```

Figure 9: Eager/Lazy Strategy Specified in the ECL

A simple modeling pointcut is shown in Figure 10. This specification binds the `EagerLazy` strategy to the data flow path starting with the "InertialSensor" component (the "start" node), and ending with the "LocDisplay" component (the "end" node). The specific parameter for the latency threshold could be changed, which would weave in different constraints into the base model. Of course, the start and end nodes of the data flow can also be changed. A more complex declaration of the starting and ending node could also be denoted, e.g., a declaration on properties of the nodes, such as whether or not a node has a publisher or consumer port.

```
aspect EagerLazyWeaponsComponents
{
  models("")->select(m | m.name() =
                         "InertialSensor")->EagerLazy("LocDisplay", 20);
}
```

Figure 10: Modeling Pointcut for Eager/Lazy Evaluation

The effect of applying the `EagerLazy` strategy across the set of modeled components can be seen in Figure 11. That figure displays the modifications made to the internals of the Update Map component. The internals of Update Map are similar to those of Compute Position, as shown in Figure 7.

In this case, a new constraint has been added (called `EagerLazy`) and the specific value of this constraint is "`assignment = Lazy`" (this can be seen in the "Constraint Equation" box in the bottom-right of Figure 11).

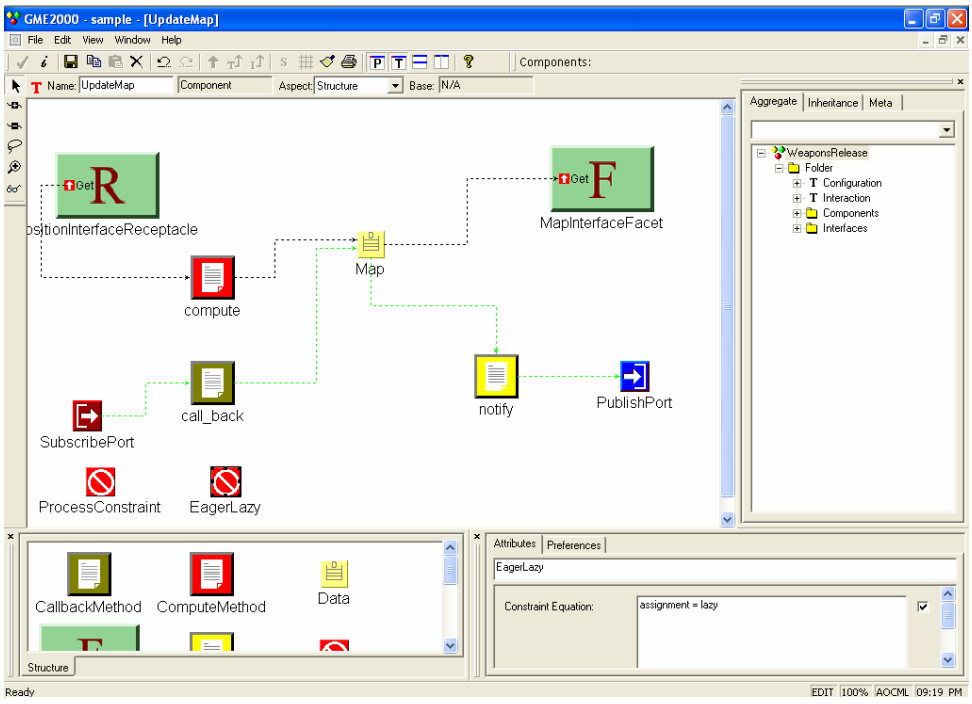


Figure 11: Effects of Eager/Lazy Strategy within Update Map Component

For application developers who are creating new instantiations of Bold Stroke, a model-based approach provides a facility for describing component configuration, assembly, and deployment information at higher-levels of ab-

straction, i.e., with visual models, as an alternative to hand-coded XML representations. Additionally, the availability of model weavers like C-SAW permits the rapid exploration of design alternatives, which are captured in constraints that specify crosscutting global properties of the modeled system.

X.3 GENERATING ASPECT CODE FROM DOMAIN-SPECIFIC MODELS

The traditional approach for generating artifacts from a domain-specific model involves the construction of an interpreter, or generator, which is then used to traverse 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. The GME provides a rich API for extracting model information.

Often, the generated artifact is represented as source code in a programming language, such as Java, C++, or C. In such cases, the interpreter has built-in knowledge of the semantics of the domain and the programming language to which it is mapped. The interpreter may also be aware of a li-

brary, 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 hard to map crosscutting 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 being *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 section is to introduce a generation technique that relies on an aspect-oriented language to encode the extended features that are added to a component.

X.3.1 Synthesizing Aspects

It is possible to generate the configuration of Bold Stroke 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 concept of "fluid" AOP, which "involves the ability to temporarily shift a program (or other software model) to a different structure to do some piece of work with it, and then shift it back" (Kiczales, 2001).

A technique for realizing this objective is the generation of AspectJ (Kiczales et al., 2001) code from models, as shown in Figure 12. In this figure, the model (top-left of figure) and modeling pointcuts (top-right of figure) are sent through a C-SAW weaver that constrains the model. Here, modeling pointcuts 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/generator 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, as illustrated by the Bold Stroke example in Section 2. 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 Service, such as the FACET work (Hunleth et al., 2001) at Washington University written using AspectJ. As an alternative, the AspectC++ weaver (Mahrenholz et al., 2002) could be used on the original C++ Bold Stroke components. For other languages, adaptations to a program transfor-

mation system, such as the Design Maintenance System (DMS, 2003), could be integrated within the model interpreter.

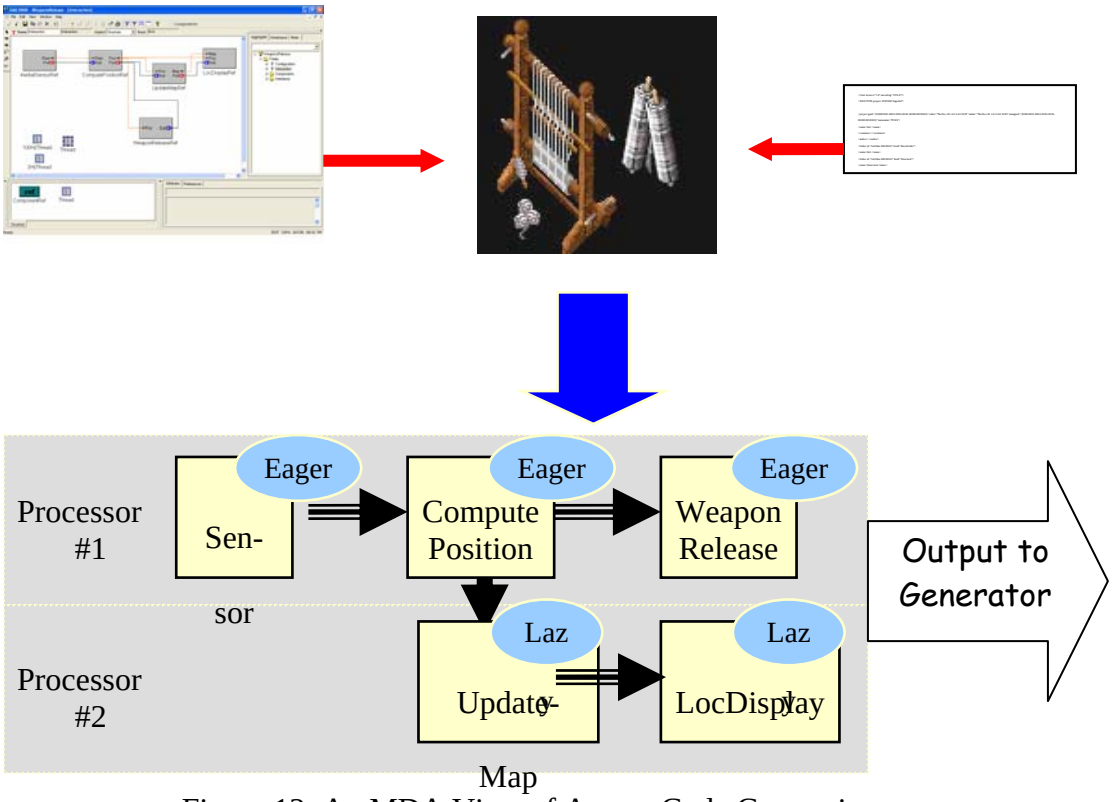


Figure 12: An MDA View of Aspect Code Generation

An example of a core library of components can be found in the Java code in Figure 13. 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 6. The `LocDisplay` subclass, for clarity, simply provides stubs for each method implementation.

```
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)

```
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 13: Base Class Java Components

Example aspects are coded in Figure 14. 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. The `Lazy` aspect exists in a library of reusable aspectual components. This abstract aspect captures the notion of lazy evaluation, as described earlier in Section 2.1.1. The callback "after" advice simply forwards all notifications to client components without making any effort to retrieve data and compute the intention of the component.

The `LocDisplayLazy` aspect, shown in Figure 14, manifests the type of code that is expected to be generated by the GME model interpreter. This code is straightforward 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. These properties are readily available to the model interpreter responsible for generating the aspect code. The code generator produces the concrete pointcuts that are needed to accomplish the weaving of the lazy evaluation concern with the `LocDisplay` component.

```
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) &&
                        execution(void LocDisplay.call_back(..));

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

}
```

b) Concretization of `Lazy` Aspect with `LocDisplay` (generated)

Figure 14: Sample Strategies and Modeling Pointcuts

To summarize the idea, it is assumed that the code shown in Figure 14a exists in a library of reusable aspects. The model synthesis step produces

code, such as that shown in Figure 14b, which represents the weaving of a particular concern as a result of model properties.

X.4 CONCLUDING REMARKS

Our work on C-SAW has demonstrated the benefits of Aspect-Oriented Software Development (AOSD) and aspect weaving at different levels of abstraction. Applying aspect-oriented techniques at the level of domain modeling is known as Aspect-Oriented Domain Modeling (AODM). AODM yields several benefits that support improvements for exploring different design alternatives in domain-specific models. When these design alternatives are captured as constraints that crosscut the modeling boundaries, separating those constraints as aspects, and then weaving them into a base model, significantly improves the capabilities for changing properties of the base model. At the implementation level, generative techniques can be used to synthesize models into executable applications. Programming language weavers, such as AspectJ, are essential for customizing pre-fabricated components from the various concerns that are described in the model.

For the AODM approach to evolve to the next level of maturity, it is necessary to have an extensive library of reusable aspects. We believe that much research is still needed, within the AOSD community in general, regarding the idea of large-scale aspect reuse and composition.

X.4.1 Future Work

The current specification of modeling aspects is done using a textual language, such as the one shown in Figure 9. Future research will be conducted on the modeling representation science required to specify model aspects using graphical formalisms. These visual representation techniques and tools will permit the specification of aspects in a manner that is consistent with the abstraction used in the specification of the base model, i.e., both aspects and models will be represented graphically within the same environment, rather than the current situation where aspects are specified textually. A common representation will also facilitate the development of a weaver that is integrated within the GME (currently, the weaving process is performed outside of the GME modeling tool, as shown in Figure 5). We are also investigating

the feasibility of using the meta-weaver with other meta-modeling tools (Gray et al., 2003).

X.4.2 Related Work

There is an increasing interest among researchers toward applying advanced separation of concerns techniques to non-code artifacts (Batory et al., 2003). In particular, AOSD techniques have been investigated at all levels of the development lifecycle, including requirements engineering (Rashid et al., 2003). Several researchers have investigated the application of AOSD concepts within the context of the UML (Clarke and Walker, 2001), (Elrad et al., 2002). These efforts have yielded guidelines for describing crosscutting concerns at higher levels of abstraction. In this regard, they have common goals with the work described in this chapter. These efforts differ from our work, however, because we have been concentrating on the idea of building actual weavers for domain models.

Within the context of distributed real-time embedded systems (DRE), the C-SAW techniques described in this chapter are being integrated within

the CoSMIC tool suite developed at Vanderbilt University (Gokhale et al., 2002). CoSMIC extends the GME to provide a modeling environment to assist in the configuration and assembly of DRE componenent middleware – in particular, systems that have QoS requirements and are assembled from components constructed using concepts from Real-Time CCM (Wang et al., 2001). Other research efforts related to CoSMIC and aspect weaving are the Virginia Embedded Systems Toolkit (VEST) from the University of Virginia (Stankovic et al, 2003), and the AIRES tool from the University of Michigan (Gu and Shin, 2003).

VEST is a toolkit that is built as a GME meta-model (Lédeczi et al., 2001). It supports modeling and analysis of real-time systems and introduces the notion of prescriptive aspects to specify programming language independent advice to a design. A distinction between VEST and our C-SAW is in the generalizabilty of the weaving process. C-SAW is constructed to work with any GME meta-model (including VEST itself), while the strength of VEST lies within real-time system specification. Additionally, the VEST prescriptive aspect language is not as rich as our ECL. According to (Stank-

ovic et al, 2003), the structure of prescriptive aspect is limited to the form, "**for** <some *conditional statement on a model property*> **change** <other *property*>," which is comparatively less powerful than the ECL capabilities demonstrated in Figure 9. The AIRES tool also has a focus on modeling of Bold Stroke scenarios. The focus of AIRES is on analysis of real-time properties, but does not adopt an aspect-oriented approach to modeling.

ACKNOWLEDGMENT

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

REFERENCES

ADR - Applied Data Research, Software Products Division Records (CBI 154), Charles Babbage Institute, University of Minnesota, Minneapolis, 2002.
(see http://www.cbi.umn.edu/collections/inv/cbi00154.html)

Batory, Don, Jacob Neal Sarvela, and Axel Rauschmeyer, "Scaling Step-Wise Refinement," *International Conference on Software Engineering*, Portland, Oregon, May 2003, pp. 187-197.

Bézivin, Jean, "From Object Composition to Model Transformation with the MDA," *Technology of Object-Oriented Languages and Systems (TOOLS)*, Santa Barbara, California, August 2001, pp. 350-354.

Booch, Grady, Ivar Jacobson, and James Rumbaugh, *The Unified Modeling Language Reference Guide*, Addison-Wesley, 1998.

Burt, Carol, Barrett Bryant, Rajeev Raje, Andrew Olson, and 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, pp. 212-223.

Clarke, Siobhán, and Robert J. Walker, "Composition Patterns: An Approach to Designing Reusable Aspects," *International Conference on Software Engineering (ICSE)*, Toronto, Ontario, Canada, May 2001, pp. 5-14.

Clements, Paul, and Linda Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.

Czarnecki, Krzysztof, and Ulrich Eiseneker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

DMS, http://www.semdesigns.com/

DOME, http://www.htc.honeywell.com/dome/

DSVL03, "Third Workshop on Domain-Specific Visual Languages," Juha-Pekka Tolvanen, Jeff Gray, and Matti Rossi, organizers, A Workshop at the Conference on *Generative Programming and*

Component Engineering (GPCE), Erfurt, Germany, September 2003, http://www.cis.uab.edu/info/GPCE-DSVL3/.

Elrad, Tzilla, Omar Aldawud, and Atef Bader, "Aspect-Oriented Modeling: Bridging the Gap Between Implementation and Design," *First ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE '02)*, Pittsburgh, PA, October 2002, pp. 189-201.

Filman, Robert, and Dan Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness," *OOPSLA Workshop on Advanced Separation of Concerns*, Minneapolis, Minnesota, October 2000.

Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns*, Addison-Wesley, 1995.

Garrett, Jason, Akos Lédeczi, and F. DeCaria, "Toward a Paradigm for Activity Modeling," *IEEE International Conference on Systems, Man, and Cybernetics*, Nashville, TN, October 2000.

Gokhale, Aniruddha, Balachandran Natarajan, Douglas C. Schmidt, Andrey Nechypurenko, Jeff Gray, Nanbor Wang, Sandeep Neema, Ted Bapty, and Jeff Parsons, "CoSMIC: An MDA Generative Tool for Distributed Real-time and Embedded Component Middleware and Applications," *OOPSLA 2002 Workshop on Generative Techniques in the Context of the Model Driven Architecture*, Seattle, WA, November 2002.

Chapter X: Two-Level Aspect Weaving to Support Evolution in Model-Driven Software

Gray, Jeff, Yuehua Lin, and Jing Zhang, "Levels of Independence in Aspect-Oriented Modeling," *Middleware 2003: Workshop on Model-driven Approaches to Middleware Applications Development*, Rio de Janeiro, Brazil, June 2003.

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

Gu, Zonghua, and Kang Shin, "A Model-Based Approach to System-Level Dependency and Real-Time Analysis of Embedded Software," *Real-Time Applications Symposium*, Washington, DC, May 2003.

Harrison, Timothy, David Levine, and Douglas C. Schmidt, "The Design and Performance of a Hard Real-Time Object Event Service," .*Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97)*, Atlanta, Georgia, October 1997, pp. 184-200.

Howard, Larry, "CAPE: A Visual Language for Courseware Authoring," *2nd OOPSLA Workshop on Domain-Specific Visual Languages*, Seattle, Washington, November 2002.

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

Karsai, Gábor, "A Configurable Visual Programming Environment: A Tool for Domain-Specific Programming," *IEEE Computer*, March 1995, pp. 36-44.

Kiczales, Gregor, 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, Gregor, "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.

Kircher, Michael, "Eager Evaluation," *European Conference on Pattern Languages of Programs*, Kloster Irsee, Germany, July 2002.

KMF, The Kent Framework, http://www.cs.ukc.ac.uk/people/staff/sjhk/kmf/

Lédeczi, Akos, 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.

Long, Earl, Amit Misra, and Janos Sztipanovits, "Increasing Productivity at Saturn," *IEEE Computer*, August 1998, pp. 35-43.

Mahrenholz, Daniel, Olaf Spinczyk, and 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, pp. 249-256.

Moore, Michael, Saeed Monemi, and Jianfeg Wang, "Integrating Information Systems in Electrical Utilities," *IEEE International Conference on Systems, Man, and Cybernetics*, Nashville, TN, October 2000.

Neema, Sandeep, Ted Bapty, Jeff Gray, and Aniruddha Gokhale, "Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems," *First ACM SIGPLAN/SIGSOFT Con-*

ference on Generative Programming and Component Engineering (GPCE '02), Pittsburgh, PA, October 6-8, 2002, pp. 236-251.

Nordstrom, Greg, Janos Sztipanovits, Gábor Karsai, and Ákos Lédeczi, "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments," *International Conference on Engineering of Computer-Based Systems (ECBS)*, Nashville, Tennessee, April 1999, pp. 68-74.

Pohjonen, Risto, and Steve Kelly, "Domain-Specific Modeling," *Dr. Dobb's Journal*, August 2002.

Polya, George, *How to Solve It*, Princeton University Press, 1957.

Rashid, Awais, Ana Moreira, and João Araújo, "Modularization and Composition of Aspectual Requirements," *2nd International Conference on Aspect-Oriented Software Development*, Boston, Massachusetts, March 2003, pp. 11-20.

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

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

Stankovic, John, Ruiqing Zhu, Ram Poornalingam, Chenyang Lu, Zhendong Yu, Marty Humphrey, Brian Ellis, "VEST: An Aspect-Based Composition Tool for Real_Time Systems," *Real-Time Applications Symposium*, Washington, DC, May 2003.

Sussman, Gerald Jay, "Robust Design through Diversity," *DARPA Amorphous Computing Workshop*, 1999.

Sztipanovits, Janos, and Gabor Karsai "Generative Programming for Embedded Systems," *First ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE '02)*, Pittsburgh, PA, October 6-8, 2002, pp. 32-49.

Tarr, Peri, Harold Ossher, William Harrison, and Stanley Sutton, "N Degrees of Separation: Multi-Dimensional Separaion of Concerns," *International Conference on Software Engineering (ICSE)*, Los Angeles, California, May 1999, pp. 107-119.

van Deursen, Arie, Paul Klint, and Joost Visser, "Domain-Specific Languages: An Annotated Bibliography," *ACM SIGPLAN Notices*, June 2000, pp. 26-36.

Vocale, Mary Lou, "JSF Virutal Battlefield," *CodeOne Magazine*, Lockheed Martin Aeronautics Company, July 2000.

Wang, Nanbor, 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, pp. 557-572.

Warmer, Jos, and Anneke Kleppe, The Object Constraint Language: Precise Modeling with UML, Addison-Wesley, 1999.