An Approach for Supporting Aspect-Oriented Domain Modeling

Jeff Gray[1], Ted Bapty[2], Sandeep Neema[2], Douglas C. Schmidt[2],
Aniruddha Gokhale[2], Balachandran Natarajan[2]

[1] Dept. of Computer and Information Sciences, University of Alabama at Birmingham
Birmingham AL 35294-1170
gray @ cis.uab.edu
http://www.gray-area.org

[2] Institute for Software Integrated Systems, Vanderbilt University
Nashville TN 37235
{bapty, sandeep, schmidt, gokhale, bala}
@isis.vuse.vanderbilt.edu
http://www.isis.vanderbilt.edu

Abstract. This paper describes a technique for improving separation of concerns at the level of domain modeling. A contribution of this new approach is the construction of support tools that facilitate the elevation of crosscutting modeling concerns to first-class constructs in a type-system. The key idea is the application of a variant of the OMG Object Constraint Language to models that are stored persistently in XML. With this approach, weavers are generated from domain-specific descriptions to assist a modeler in exploring various alternative modeling scenarios. The paper examines several facets of Aspect-Oriented Domain Modeling (AODM), including: domain-specific model weavers, a language to support the concern separation, an overview of code generation issues within a meta-weaver framework, and a comparison between AODM and AOP. An example of the approach is provided, as well as a description of several future concepts for extending the flexibility within AODM.

1 Introduction

The benefits of performing refinements on non-code artifacts are well documented [3]. Our contribution to this area has been in Aspect-Oriented Domain Modeling (AODM), which represents the union of Aspect-Oriented Software Development (AOSD) [1] and Model-Integrated Computing (MIC) [34]. An AOSD approach can be beneficial at different stages of the software lifecycle and at various levels of abstraction. In particular, it can be advantageous to apply AOSD principles at levels closer to the *problem space*,

e.g., architectural analysis [18], requirements engineering [30], and modeling [14], as well as the *solution space*, e.g, design [5, 10, 33], and implementation/coding [4, 19, 21, 36].

The advantages of applying AOSD to domain modeling are considerable. AODM assists a modeler in capturing concerns that were previously hard, if not impossible, to modularize (see the introductory example in Section 3). A key benefit is the ability to explore numerous scenarios by considering crosscutting modeling concerns, such as desired fault tolerance or latency levels, as aspects that can be inserted and removed from a model rapidly.

A growing area of research is concentrated on bringing aspect-oriented techniques into the purview of analysis and design (see [5, 10, 33] for examples of work in this area). A focal point of these efforts is the development of notational conventions that assist in the documentation of concerns that crosscut a design. These notational conventions advance the efficiency of expression of these concerns in the design. Moreover, they also have the important trait of improving the traceability from design to implementation. Although these current efforts do well to improve the cognizance of AOSD at the design level, they generally tend to treat the concept of aspect-oriented design primarily as a specification convention. This is to say that the focus has been on the graphical representation, semantical underpinnings, and decorative attributes concerned with aspects and their representation within UML. A contribution of this paper is to consider AODM more as an operational task by constructing executable model weavers. That is, we view AOSD as a mechanism to improve the modeling task itself by providing the ability to quantify properties across a model *during* the system modeling process. This action is performed by utilizing a weaver that has been constructed with the concepts of domain modeling in mind. A research effort that also appears to have this goal in mind can be found in [17], although this work seems more aimed at providing a transformation tool that reifies patterns at the level of object-oriented design.

The successful application of AODM necessitates the availability of weavers that understand the underlying modeling domain. These weavers process *models*, not source code, so programming language compilers like AspectJ [19] are not applicable due to the semantic mismatch of the abstraction level. Because the syntax and semantics of each modeling domain are unique, a different weaver is needed for each domain. To support this requirement, we have developed a meta-weaver framework to assist in the creation of new model weavers. We call this framework the Constraint-Specification Aspect Weaver (C-SAW) – (Note: the name is borrowed from the realization that a crosscutting saw, or c-saw, cuts across the grain of wood). This framework uses several code generators whose inputs are meta-level specifications, described in a Domain-Specific Language (DSL), which hide accidental complexities of interacting with XML and COM. The generators produce code that is merged into the C-SAW framework to instantiate a domain-specific weaver.

The remainder of this introduction provides the background information needed to understand the modeling context for the emergence of scattered constraints.

1.1 Model-Integrated Computing

Expressive power in software specification is often gained from using notations and abstractions that are aligned with the problem domain. This can be further enhanced when graphical representations are provided to model the domain abstractions. In our particular approach to domain-specific modeling, a design engineer describes a system by constructing a visual model using the terminology and concepts from a specific domain. Analysis can then be performed on the model, or the model can be synthesized into an implementation [20, 24, 34, 35].

Model-Integrated Computing (MIC) has been refined over many years to assist in the creation and synthesis of complex computer-based systems. A key application area for MIC is in those systems that have a tight integration between the computational structure of a system and its physical configuration (e.g., embedded systems) [34]. In such systems, MIC has been shown to be a powerful tool for providing adaptability in changing environments [35].

The Generic Modeling Environment (GME) [20] is a meta-configurable modeling environment for realizing the principles of MIC. The GME provides meta-modeling capabilities that can be configured and adapted from meta-level specifications (representing the *modeling paradigm*) that describe the domain. There are several domains to which MIC and the GME have been successfully applied. The most notable evidence for the advantages of applying model-driven techniques is found in [22], where the documented benefits are described from the initiation of MIC into an automotive factory process.

1.2 Design Space Exploration in a Product-Line Architecture

A beneficial approach toward domain modeling considers the creation of a base model for representing a family of related systems, often called product-line architecture [6]. In such an approach, a design space corresponds to a set of implementation alternatives that are available within the product family. The selection of a fixed-point, among the set of possible alternatives from the base model, must be explored prior to model synthesis [23]. Design space exploration is an iterative process that selectively evaluates a set of constraints that are chosen by a modeler using a tool.

The exploration of a design space often requires the existence of constraints that are dispersed throughout a model [23]. Constraints codify properties of the model that must be satisfied during exploration. A modeler can specify constraints in the GME as model attributes that are then evaluated during design-space exploration. An example of a constraint is an assertion about the end-to-end latency within the flow of a sub-model. Each iteration of the exploration prunes the design space further. Focusing the exploration on different sets of constraints can lead the exploration and pruning algorithms along different elaborations of synthesis.

Although constraints are a necessary modeling construct for supporting design space exploration, the next section explains why constraints emerge as a crosscutting modeling entity.

2 Model Weavers for Separating Crosscutting Constraints

The primary goal of AOSD is to assist in modularizing crosscutting behavior [4, 19, 21, 36]. In the same manner that crosscutting code detracts from the cohesiveness of an implementation, the utility of specifying constraints within a model is often diminished due to their scattering throughout the model hierarchy [14]. It is often the case that the meta-model forces the emergence of a "dominant decomposition" (i.e., the primordial criteria for modular decomposition) [8, 36] that imposes the subjugation of other concerns, such as those captured by constraints.

In conventional system modeling tools, any change to the intention of a global property requires visiting and modifying each constraint, for every context, representing the property. This requires the modeler to "drill-down" (i.e., traverse the hierarchy by recursively opening, with the mouse, each sub-model), manually, to many locations of the model. It is common for a model in the GME to contain thousands of different modeling elements with hierarchies that are ten or more levels deep. The interdependent nature of each constraint makes change maintenance a daunting task for anything but a simple model. The benefits of a single model representation of a product family are nullified because the "Parnasian" objectives [27] of changeability, comprehensibility, and independent development are sacrificed in the presence of crosscutting constraints.

As models grow in size and complexity, it becomes unmanageable to view the contents of a model in its entirety because there are too many participating entities. The concept of viewpoints has been researched frequently as a topic within requirements engineering [25]. The GME supports the concept of a viewpoint as a first-class modeling construct, which assists a modeler in separating the concerns of multi-perspective views [20]. Each GME viewpoint describes a partitioning that selects a subset of entities as being visible.

Although they offer a powerful conceptualization for concern separation, GME's implementation of viewpoints, however, does not fit completely within the definition of aspects (at least in the way that they are defined within the AOSD community). Using only viewpoints, for example, a modeler cannot quantify over a model's join points and apply advice. The key parts of AOP, as enumerated in [19], are not fully present in many viewpoint-oriented implementations. Research into aspectual requirements also suggests that viewpoints alone are incapable of capturing many crosscutting concerns [30].

Because the current viewpoint implementation in most modeling tools does not adequately capture crosscutting concerns, a new extension to modeling tools is needed. We further motivate this need in the next section and provide an introduction to our approach for AODM.

2.1 The Need for Domain-Specific Model Weavers

Different domains typically will have different dominant decompositions and dissimilar crosscutting concerns. For instance, the adaptation of the frame rate or size of a visual display in an avionics system would have no counterpart in a domain that models an automotive factory. Consequently, because each new GME meta-modeling paradigm introduces different types of modeling elements, syntax, and semantics, different weavers are needed for each new modeling paradigm. The situation is similar to the reason that a different compiler is needed for a new programming language – the syntax and semantics typically vary too much between each language to permit a single instance of a generalized translator that compiles multiple languages. Thus, the domain for automotive manufacturing (e.g., a Saturn car) [22] needs its own specialized weaver, as does the BBN Unmanned Aerial Vehicle (UAV) domain [31], and the Boeing Bold Stroke domain [32].

```
<model id="id-05" kind="Component">
 <name>InertialSensor</name>
 <atom id="id-17" kind="ComputeMethod" role="ComputeMethod">
  <name>compute</name>
    <attribute kind="WCET">
      <value>2</value>
    </attribute>
```

Fig. 1. Bold Stroke XML Model

```
<model id="id—544975-39" kind="State">
  <name>frameRate</name>
<model id="id—544975-42" kind="State">
  <name>Range1-7</name>
<connection id="id—544975-63" kind="Transition">
  <name>Transition</name>
<connpoint role="dst" target="id—544975-42" />
<connpoint role="src" target="id—544975-46" />
  <attribute kind="Guard">
    <value>latency > 25</value>
  </attribute>
  <attribute kind="Action">
    <value>frameRate=4</value>
  </attribute>
```

Fig. 2. BBN/UAV XML Model

The GME has the capability to store models persistently using XML. To better understand the need for multiple weavers, consider the XML document in Figure 1. This represents a subset of a domain model description. The document has distinctly named regions with respect to the kind of elements being presented in the domain (e.g.,

"Component"), as well as roles (e.g., "ComputeMethod"), name, and even attributes (e.g., "WCET"). This is the meta-description of the Bold Stroke domain.

Further consider the XML fragment in Figure 2. It also has its own unique modeling entities (e.g., "State," "Transition," "Guard"). It should be noted that the same XML DTD is used in both Figures 1 and 2. However, the modeling concepts captured in each model are significantly different. The quoted strings in some of these models (e.g., the "kind" slots) show that something "meta" is truly happening.

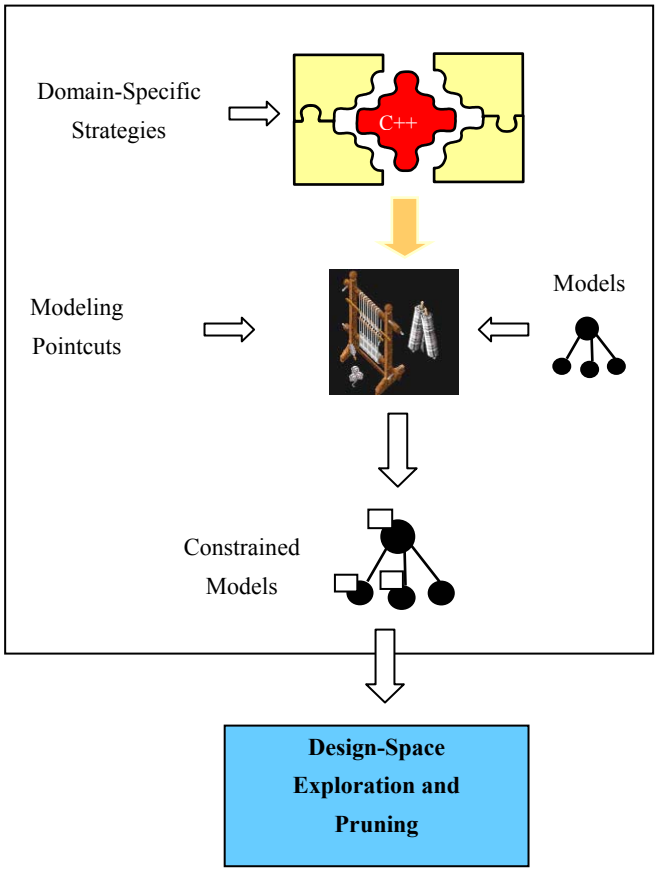

Fig. 3. Summary of AODM Process

Because of the diversity of domains, the ability to construct weavers for new domains is desired. The AODM approach that we are using can be summarized by the diagram in Figure 3. In this figure, new weavers are created by integrating domain-specific strategies

into a meta-weaver framework (shown in the top-part of Figure 3). A *strategy* specifies a heuristic (e.g., processor assignment, as shown in the example in Section 3) for a specific modeling paradigm. Strategies are specified in a DSL called the Embedded Constraint Language (ECL), which is described in Section 4. A generator translates each strategy into C++, such that an instantiation of the meta-weaver framework is created (i.e., the generated C++ is in the middle of the framework). The instantiation of the framework (with a set of strategies) produces a new domain-specific weaver (middle of Figure 3). After a weaver is created for a specific domain, GME models (represented in that domain) can be woven with modeling pointcuts. A *modeling pointcut* identifies specific points in a model that are affected by a crosscutting modeling concern.

As mentioned earlier, the output of a domain-specific weaver is a new GME model that contains constraints that have been woven, i.e., the input to the weaver may be a base model that is void of any constraints, like the middle-right of Figure 3. The newly created constrained model can then be passed on to the design-space exploration tool, as mentioned previously in Section 1.2. The content inside the box of Figure 3 represents our contributions to AODM. The design space exploration research is a previous effort that provided the initial motivation for exploring this new area.

3 Example

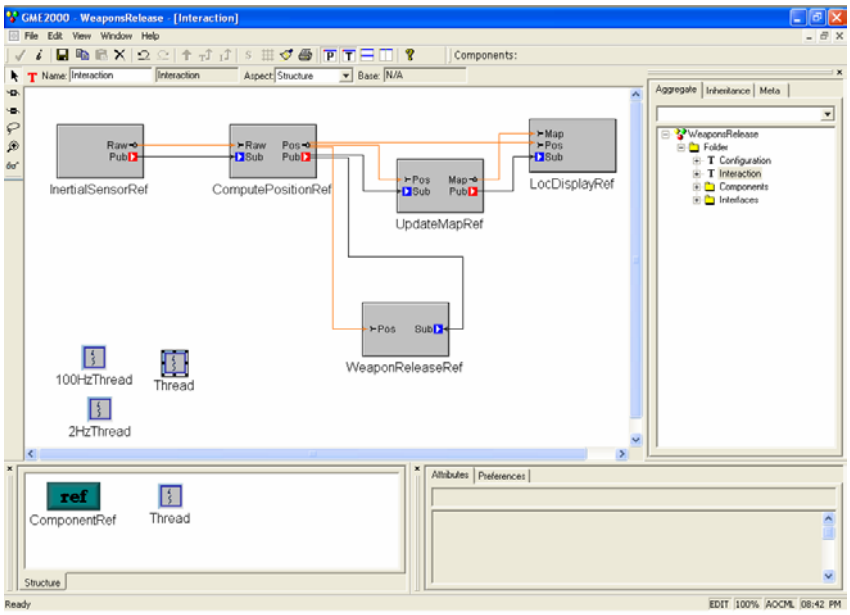

Fig. 4. Bold Stroke Component Interactions as Modeled in the GME

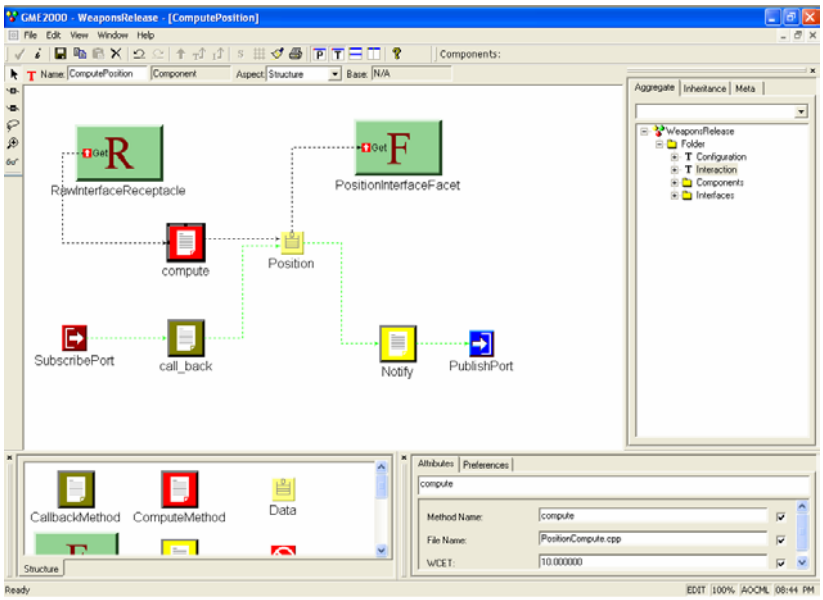

Fig. 5. A GME Model of the Internals of `Compute Position`

Bold Stroke is a product-line framework from Boeing for avionics navigation software [32]. In this section, an example crosscutting modeling concern is presented in a domain for modeling a subset of Bold Stroke applications and configurations.

Consider the requirements for a simple model that contains five software components representing a simplified scenario of an avionics mission program (see Figure 4). The first component is an inertial sensor. This sensor outputs, at a 100Hz rate, 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. It has a fixed period of 2Hz and a minimal-latency requirement. 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. Each of these components has distinct frequencies, latencies, and Worst Case Execution Times (WCET) [29]. The specific values of these properties will likely differ depending on the type of aircraft represented by the model, e.g., the latencies and WCETs for an F/A-18 fighter aircraft 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 4 depicts the weapons deployment model represented within the GME. The model is an instance of the domain that was developed initially for modeling of Bold Stroke applications and component-based middleware. Each of the components in Figure 4 has internal details that also are modeled. For instance, the contents of the "Compute Position" component are rendered in Figure 5. As shown in the internals of this component, the series of interactions actually take place using a publish/subscribe model. The figure specifically highlights the attributes of a method called "compute" (see the bottom-right of the figure). The attributes provide the name of the method, the C++ source file that contains the method, and the method's estimated WCET.

3.1 Example Crosscutting Concern: Processor Assignment

Suppose that we wanted to model the processor assignment of each component. That is, based upon the expected WCET, the component methods are executed as tasks on various processors. A notation is needed to specify the assignment of component methods/tasks to processors. One way to accomplish this representation issue is to specify the processor assignment as a constraint of the component model.

The way that processor assignment is typically modeled involves the application of a set of heuristics that globally assign tasks to processors based on specific properties of each component. In modeling, this often requires the modeler to visit each component, or task, in order to manually apply the heuristic. For a model with a large number of components, this can be a daunting task. It becomes increasingly unmanageable in situations where the modeler would like to play "what-if" scenarios. These "what-if" scenarios are used to drive the iterative evolution of the model, such that intermediate scenarios may even be discarded. This is helpful because a modeler may want to change the values of different properties, or even modify the details of the heuristic, in order to observe the effect of different scenarios. A manual application of a heuristic would require that the modeler re-visit every component and re-apply the rules of the heuristic.

An example of our approach for separating the concern of processor assignment can be found in Figures 6 and 7. The details of the language are defined elsewhere in the paper, but an outline of the meaning of these figures is offered here. The interpretation of the pointcut called `ProcessorAssignment` (Figure 7) is that a selection is specified over all of the modeling elements that are of type "Comp*" (note the use of the wildcard designator). Although not shown here, modeling pointcuts can also be formally named and composed with other pointcuts. It is not necessary that a pointcut be bound to a strategy, but the pointcut in Figure 7 is tied to a particular strategy called `Assign` (Figure 6). The combination of the pointcut and strategy invokes `Assign` on each of these modeling components (here, a parameter bound to the value 10 represents a threshold of the execution time for each processor load). The purpose of the `Assign` strategy is to look into the "compute" method of each component and find its WCET. The WCETs of each component are accumulated. Whenever this accumulated value reaches past the threshold, a new processor is created for component assignment. `Assign` will finally call

another strategy, named `AddConstraint`, which will add a new constraint to the model. The new constraint, in this case, represents the processor assignment.

Note that the `ProcessorAssignment` pointcut could be modified so that a different strategy is invoked (i.e., some strategy other than `Assign);` or, a different parameter threshold could be provided that may result in a different set of constraints (i.e., the parameter to `Assign` may be changed from 10 to 20). The key advantage of this approach is realized in the observation that, from a change in *one* place, an entirely different set of constraints can be weaved into the model. This solves a serious scalability problem concerning maintenance issues, and the ability to change the constraints within a model.

```
defines AddConstraint, Assign, ProcessorAssignment;

strategy AddConstraint(constraintName, expression : string)
{
  addAtom("OCLConstraint", "Constraint",
          constraintName).addAttribute("Expression", expression);
}

strategy Assign(limit : integer)
{
  declare static accumulateWCET, processNum : integer;
  declare currentWCET : integer;
  declare aConstraint : string;

  self.compute.WCET.getInt(currentWCET);
  accumulateWCET := accumulateWCET + currentWCET;

  if (limit < accumulateWCET) then
    accumulateWCET := currentWCET;
    processNum := processNum + 1;
  endif;

  aConstraint = "self.assignTo() = processor" + processNum;
  AddConstraint("ProcessConstraint", aConstraint);
}
```

Fig. 6. Strategy for Processor Assignment

```
pointcut ProcessorAssignment
{
  models("")->select(m | m.kind() = "Comp*")->Assign(10);
}
```

Fig. 7. Pointcut Defining Model Locations for Applying the `Assign` Strategy

In comparison to the weaving performed at the coding level, as typified by AspectJ, the pointcut specification is encapsulated with the advice in order to describe where and when the aspect is to be applied. We took a different approach in the mechanism for specifying crosscutting modeling concerns. In our approach, the pointcut and strategies are often specified in separate files. This permits better reuse among the pointcuts and strategies (i.e., the pointcuts are more transparent to the individual strategy definitions).

Figure 8 shows the same component that was given in Figure 5. The only difference is that the component now contains a constraint that was added by the weaver as a result of applying the strategies described by the pointcut. Notice that the strategy has assigned this component to processor 0. An examination of all the other components involved in this interaction would reveal that different components are assigned to processors based on their WCET and the parameterized threshold.

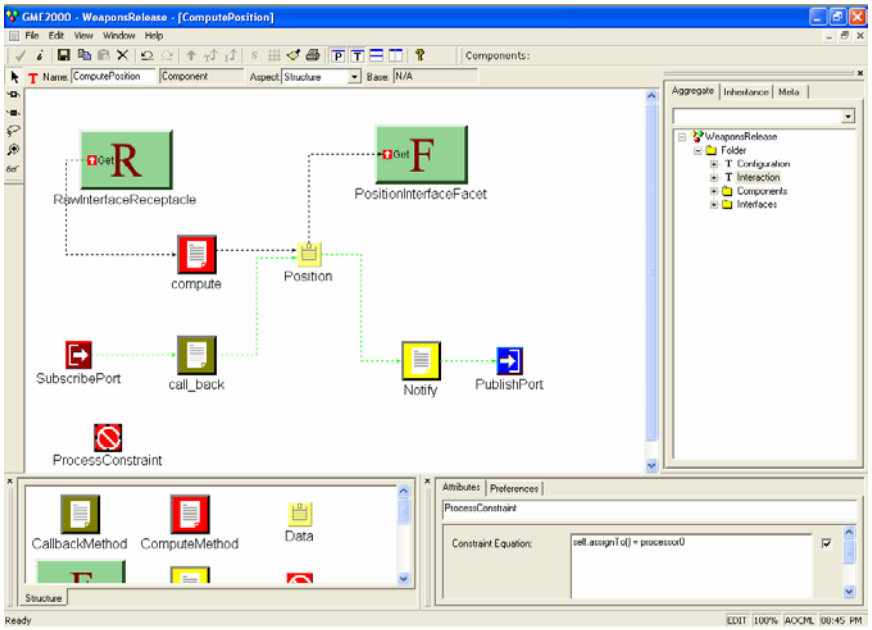

Fig. 8. Component with Weaved Constraint

4 Embedded Constraint Language

Model weavers are specified using the ECL - an extension (and subset) of the OMG Object Constraint Language (OCL) [37]. This language allows the weaver designer to specify the traversal of models, computations upon the model structure and attributes, and subsequent modifications to the models. In essence, the ECL is used to describe the transformations of an existing domain model that are needed to represent the crosscutting modeling concerns. Examples of the type of modifications that can be performed on models would be the addition of constraint objects, addition/modification of attributes to existing models, and addition of domain-specific modeling objects. A short description of the ECL follows.

Table 1. Included OCL Operators

Arithmetic Operators
`+, -, *, /, =, <, >, <=, >=, <>`
Logical Operators
`and, or, xor, not, implies, if/then/else`
Collection Operator & Property Operator
`->`
`.`
Standard OCL Collection Operators
`collection->size() : integer`
`collection->forAll(x | f(x) ) : Boolean`
`collection->select(x | f(x) ) : collection`
`collection->exists(x | f(x) ) : Boolean`

The ECL supports many of the basic language constructs found in the OCL, as categorized in Table 1. The following capabilities distinguish ECL from OCL:

- ECL provides a set of operators to navigate the hierarchical structure of a model (see Table 2). These aggregate 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 modeling pointcut, such as `findModel`, `getID`, `findAttribute`. These operators are akin to the introspective operators in Java (e.g., `getName`, `getType`, `getInt`); i.e., they are reflective to the

internal representation used in the GME. These operators, and the standard OCL selection operators, have similarities to the submitted OMG proposals to support Query/View/Transformations (QVT) [26] (e.g., CompuWare's TPL, and Rational XDE's pattern engine). In ECL, a query across the model can be specified using these navigational operators. The underlying XML representation of the model is searched by translating the ECL navigational statements into the XPath querying language.

- ECL also supports the "Transformation" idea of the OMG QVT. Traditionally, OCL has been used as a declarative language to specify properties of UML diagrams [37]. The use of ECL, however, requires the capability to introduce side-effects into the underlying XML model. This capability is needed because the strategies often specify transformations that must be performed on the model, which requires the ability to make modifications to the model as the strategy is applied. ECL therefore supports an imperative procedural style with numerous operations that can alter the state of the model, such as `addAtom`, `addAttribute`, `removeChild`. Because the underlying model hierarchy is stored as an XML file, these functions are often implemented as wrappers for the specific calls that are needed to use XPath and the XML Document Object Model (DOM).
- The procedural nature of ECL permits dependencies between strategies. Strategies can be chained together as procedure calls. Recursion is also supported in the ECL. Circular dependencies are possible (of course, the strategy must specify a termination condition for the strategy to complete its processing).

Table 2. ECL Model Operators

Aggregates
`folders, models, atoms, attributes, connections`
Connections
`connpoint, target, refs, resolveReferredID`
Transformation
`addAttribute, addAtom, addModel, addConnection, removeNode`
Selection
`findFolder, findModel, findAtom, findAttributeNode`
General
`id, parent, getID, getInt, getStr`

5 Comparison to AOP

Domain-specific weavers rely on modeling pointcuts and strategies to perform their responsibilities. Modeling pointcuts are used to describe *where* the concern will be applied in the model, and strategies describe *how* a concern is applied in the context of a particular node in the model. Several comparisons can be made between the approach to AODM described in this paper and traditional AOP.

Table 3 provides a comparison of the critical elements that make a system aspect-oriented, according to the definition provided in [19]; i.e., the join point model, the pointcut designator construct, and the concept of advice. The AODM approach presents a way to query and traverse over a large model space. As such, the approach has borrowed from the experience of traversal specifications as typified by work done in Demeter and Adaptive Programming techniques [21]. The crucial difference is that the implementations of Demeter have primarily focused on code-level traversals. Our models are graphical representations of a domain at a higher level of abstraction, thus necessitating a different focus.

Table 3. Comparison of AspectJ and AODM

	AspectJ	AODM
Join Point Model	Well-defined points in the execution of a program	Static points (nodes) in a model
Pointcut Designator	A declarative statement (formed from a set of primitives like `call`, `this`, and `target`) that describes a set of join points in a program	A declarative statement (formed from ECL collection operators) that identifies a set of locations within a model
Advice	A block of code that is executed at a join point	A strategy, or heuristic, for instrumenting a model with information related to a concern

The box in the bottom of Figure 9 represents a subset of a modeling pointcut. In the pointcut of Figure 9, a predicate within the `select` statement instructs the weaver to collect all nodes in the model that are of kind “StateFlow” and have a name that matches “Model*.” Such a statement has a direct correspondence to a pointcut (as in AspectJ) that picks out specific points in the execution of a program satisfying some condition. The modeling pointcut also describes the strategy that is to be invoked on each node selected from the predicate. The effect of this association is the quantification of a concern over multiple join points [11]. As the strategy is applied at each node, the model is transformed according to the intent of the strategy, which has a direct correspondence to the

association of pointcuts with advice in AspectJ, and how advice affects the execution of the program (of course, our models as represented in XML are static).

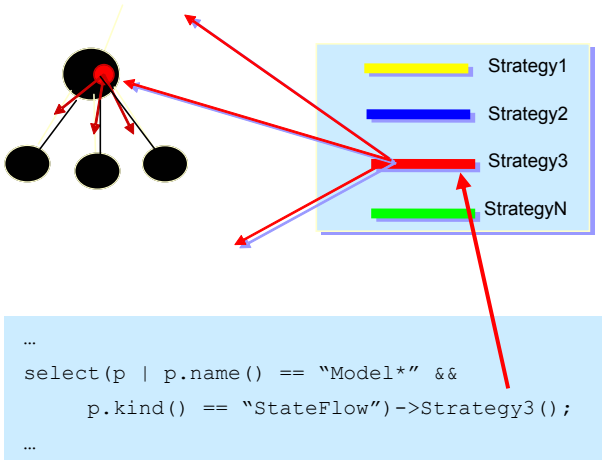

Fig. 9. Effects of the Coordination Between Modeling Pointcuts and Strategies

6 Code Generation

This section discusses issues related to the development of the ECL code generator. In particular, the benefits of using a domain-specific language (DSL) to isolate several accidental complexities (e.g., the lower-level XML DOM, and the COM data structures) are described.

The Strategy Code Generator (StratGen) tool translates strategies, as specified in the ECL, into C++ code that can be inserted into the meta-weaver framework (see the top of Figure 3). This sub-section provides an example of the translation approach used within StratGen.

```
components.models("")->select(c |
            c.id()==refID)->DetermineLaziness();
```

Fig. 10. Fragment of an EagerLazy Strategy

Figure 10 contains a statement from a strategy, described in [16], which is focused on eager/lazy evaluation for a CORBA event channel (this is just one of several lines found in that strategy – it is not meant to imply that this single line represents the entirety of the strategy). This statement finds all of the models that match a specific id and then calls the

`DetermineLaziness` strategy on those selected nodes. The amount of C++ code that is generated by StratGen, however, is far from being concise or simple (see Figure 11). Much of the code for implementing this strategy statement is focused on iterating over a collection and selecting elements of the collection that satisfy the predicate. The ECL hides the accidental details that pertain to the fact that the underlying model is represented in XML. As such, the COM invocations of the lower-lever API calls to manipulate the XML DOM are concealed.

```
…
CComPtr<IXMLDOMNodeList> models0 = XMLParser::models(components, "");
nodeTypeVector selectVec1 = XMLParser::ConvertDomList(models0);
nodeTypeVector selectVecTrue1 = new std::vector<nodeType>;
vector<nodeType>::iterator itrSelect1;
for(itrSelect1 = selectVec1->begin();
    itrSelect1 != selectVec1->end(); itrSelect1++) {
  nodeType selectNode1 = (*itrSelect1);
  nodeType c;
  c = selectNode1;
  CComBSTR id0 = XMLParser::id(c);

  ClData varforward1(id0);
  ClData varforward2(referredID);
  bool varforward3 = varforward1 == varforward2;
  if(varforward3)
     selectVecTrue1->push_back(*itrSelect1);
}

vector<nodeType>::iterator itrCollCall1;
for(itrCollCall1 = selectVecTrue1->begin();
    itrCollCall1 != selectVecTrue1->end(); itrCollCall1++)
  DetermineLaziness::apply(…);
…
```

Fig. 11. Sample of Generated C++ Code

The code in Figure 11 contains a generic value class named `ClData`. It is in this class where the equality operator performs a special match for string wildcards. The C++ code calls an XML Parser wrapper class that retrieves a set of all models. An iteration over the list of models checks to see if the name of the node referenced by the current iterator matches the wildcard. The ECL was one of several candidate languages used in a study of the conciseness of DSLs [15]. In that study, the ECL was shown to be 3 times more concise than the representative C++.

7 Future Work

The ECL has truly been an evolving language – each new strategy that was created brought fresh insight into additional language constructs that would be beneficial. In the future, the ECL will continue to evolve to support additional features (e.g., support for a

"cflow" or "dflow" modeling construct, similar to AspectJ). This section outlines some additional research objectives that will be explored in the immediate future.

A potentially rewarding subject for future investigation will be to subsume the textual descriptions formulated within the ECL into a graphical modeling language. This effort will investigate the expression of modeling pointcuts, and even strategies, using a graphical formalism similar to that of visual programming languages. This kind of visual aspect modeling would, of course, be perfectly suited for exploration from within the GME. The concept of generating weavers from visual formalisms (i.e., interpreting strategy specifications that are described visually) is also appealing.

The current version of C-SAW assumes that the separation of modeling concerns was being performed on models created with the GME. In fact, this assumption is built into the XML Parser within the weaver framework. The limitation imposed by this assumption precludes other modeling tools (that also can export models using XML) from being able to employ the benefits of an aspect weaver. In addition to the GME, other examples of domain-specific visual modeling tools are Honeywell's Domain Modeling Environment [9], and metaEdit+ (from metaCASE) [28]. It is possible that these, and other modeling tools could benefit from an aspect-oriented modeling approach. A new code generator could be inserted into the weaver framework in order to provide an added measure of variability. From the modeling tool's Document Type Definition (DTD), the functionality of the wrappers provided within the XML Parser can be generated. This would permit adaptability of the framework between domains (using the strategy code generator), and also adaptability between modeling tools, using Generative Programming [7] and invasive composition techniques [2].

A future goal of our project is to provide the capability for generating 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. For example, a base model can capture the infrastructure of a product-line with constraints representing specific configuration information for a particular product (e.g., for distributed real-time embedded systems [13]). A synthesis process can generate AspectJ components from an analysis of the model and constraints (initial ideas for supporting this have been presented in [16]). This goal fits well with quality of service issues applied to the OMG's Model Driven Architecture (MDA) [12].

8 Concluding Remarks

The main objective of the research described in this paper is to apply the concepts of AOSD to domain modeling. The implementation of this objective has resulted in a means to add aspect modeling to the repertoire of the well-established GME modeling tool. The result of our work is a model weaver framework called the Constraint-Specification Aspect Weaver (C-SAW). Earlier work on aspect modeling has concentrated on important notational issues for extending the UML, whereas the research described in this paper has brought the benefits of aspect-orientation to the modeling process itself. The work

described in this paper has been applied to modeling efforts of Boeing Bold Stroke [16, 32]. A model weaver has also been demonstrated with BBN's adaptive UAV project [31], as briefly described in [24].

There are several reasons that would support the adoption of our approach into a general modeling paradigm. It has been discovered that a lack of support for separation of concerns with respect to constraints can pose a difficulty when creating domain-specific models. Constraints may be specified throughout the nodes of a model to stipulate design criteria and limit design alternatives. However, because these constraints are scattered across the hierarchy of a model, they are hard to change. The scattering of constraints throughout various levels of a model makes it hard to maintain and reason about their effects and purpose.

The concept of a domain-specific weaver can be used in many ways beyond the application of constraints. For example, a weaver can be used to distribute any system property endemic to a specific domain across the hierarchy of a model. A weaver can also be used to instrument structural changes within the model according to the dictates of some higher-level requirement that represents a crosscutting concern.

The C-SAW weaver framework serves as a generalized transformation engine for manipulating models. The framework, in conjunction with several code generators and DSLs, is used to provide the adaptability needed to construct new instances of the framework. A core component of this framework is a code generator that translates high-level descriptions of strategies into C++ source code. The conciseness of the ECL, compared to the generated code, provides a measure of the benefit for using DSLs to provide a higher level of abstraction.

Acknowledgements

We would like to thank the following individuals for comments on portions of this paper: Steve Schach, Gábor Karsai, Janos Sztipanovits, and Ákos Lédeczi. Also, discussions with Dave Sharp and other Boeing PCES members have been essential toward improving our understanding of the role of modeling in the development of Bold Stroke. Several comments provided by the anonymous reviewers were beneficial in improving the clarity of the paper.

The DARPA Information Exploitation Office (DARPA/IXO), under the Program Composition for Embedded Systems (PCES) program, funds this work.

References

1. http://www.aosd.net
2. Uwe Aßmann, *Invasive Software Composition*, Springer-Verlag, 2003.
3. Don Batory, Jacob Neal Sarvela, and Axel Rauschmeyer, "Scaling Step-Wise Refinement," *International Conference on Software Engineering*, Portland, Oregon, May 2003, pp. 187-197.

4. Lodewijk Bergmans and Mehmet Aksit, "Composing Crosscutting Concerns using Composition Filters," *Communications of the ACM*, October 2001, pp. 51-57.
5. Siobhán Clarke and Robert J. Walker, "Towards a Standard Design Language for AOSD," *First International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 2002, pp. 113-119.
6. Paul Clements and Linda Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.
7. Krzysztof Czarnecki and Ulrich Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.
8. Maja D'Hondt and Theo D'Hondt, "The Tyranny of the Dominant Model Decomposition," *OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture*, Seattle, Washington, November 2002.
9. http://www.htc.honeywell.com/dome/
10. Tzilla Elrad, Omar Aldawud, Atef Bader, "Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design," *Generative Programming and Component Engineering (GPCE),* Pittsburgh, Pennsylvania, October 2002, pp. 189-201.
11. Robert Filman and Dan Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness," *OOPSLA Workshop on Advanced Separation of Concerns*, Minneapolis, Minnesota, October 2000.
12. David Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons, 2003.
13. Aniruddha Gokhale, Douglas Schmidt, Balachandran Natarajan, Jeff Gray, and Nanbor Wang, "Model-Driven Middleware," in *Middleware for Communications*, (Qusay Mahmoud, ed.), John Wiley & Sons, 2003.
14. 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.
15. Jeff Gray and Gábor Karsai, "An Examination of DSLs for Concisely Representing Model Traversals and Transformations," *36th Hawaiian International Conference on System Sciences (HICSS),* Big Island, Hawaii, January 6-9, 2003.
16. Jeff Gray, Janos Sztipanovits, Douglas C. Schmidt, Ted Bapty, Sandeep Neema, and Aniruddha Gokhale, "Two-level Aspect Weaving to Support Evolution of Model-Based Software," in *Aspect-Oriented Software Development*, (Robert Filman, Tzilla Elrad, Mehmet Aksit, and Siobhán Clarke, eds.), Addison-Wesley, 2003.
17. Wai-Meng Ho, Jean-Marc Jezequel, Francois Pennaneac'h, and Noel Plouzeau, "A Toolkit for Weaving Aspect-Oriented UML Designs," *First International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 2002, pp. 99-105.
18. Mika Katara and Shmuel Katz, "Architectural Views of Aspects," *2nd International Conference on Aspect-Oriented Software Development*, Boston, Massachusetts, March 2003, pp. 1-10.
19. 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.
20. Ákos 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.
21. Karl Lieberherr, Doug Orleans, and Johan Ovlinger, "Aspect-Oriented Programming with Adaptive Methods," *Communications of the ACM*, October 2001, pp. 39-41.

22. Earl Long, Amit Misra, and Janos Sztipanovits, "Increasing Productivity at Saturn," *IEEE Computer*, August 1998, pp. 35-43.
23. Sandeep Neema, "System Level Synthesis of Adaptive Computing Systems," Ph.D. Dissertation, Vanderbilt University, Dept. of Electrical Engineering and Computer Science, May 2001 (http://www.isis.vanderbilt.edu/publications/archive/Neema_S_5_0_2001_System_Lev.pdf)
24. Sandeep Neema, Ted Bapty, Jeff Gray, and Aniruddha Gokhale, "Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems," *Generative Programming and Component Engineering (GPCE)*, Pittsburgh, Pennsylvania, October 2002, pp. 236-251.
25. Basher Nuseibeh, Jeff Kramer, and Anthony Finkelstein, "A Framework for Expressing the Relationship Between Multiple Views in Requirements Specification," *IEEE Transactions on Software Engineering,* October 1994, pp. 760-773.
26. *OMG Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*, OMG Document: ad/02-04-10, April 2002.
27. David Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, December 1972, pp. 1053-1058.
28. Risto Pohjonen and Steve Kelly, "Domain-Specific Modeling," *Dr. Dobb's Journal*, August 2002.
29. Peter Puschner and Alan Burns, "A Review of Worst-Case Execution Time Analysis," *The Journal of Real-Time Systems*, Vol. 18, Number 2/3, pp. 115-128.
30. Awais Rashid, 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.
31. Richard Schantz, Joseph Loyall, Michael Atighetchi, and Partha Pal, "Packaging Quality of Service Control Behaviors for Reuse," *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC),* April/May 2002, Washington, D.C., pp. 375-385.
32. David Sharp, "Reducing Avionics Software Cost Through Component Based Product-Line Development," *Software Technology Conference*, Salt Lake City, Utah, April 1998.
33. Dominik Stein, Stefan Hanenberg, and Rainer Unland, "An UML-based Aspect-Oriented Design Notation," *First International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 2002, pp. 106-112.
34. Janos Sztipanovits and Gábor Karsai, "Model-Integrated Computing," *IEEE Computer*, April 1997, pp. 10-12.
35. Janos Sztipanovits, "Generative Programming for Embedded Systems," *Keynote Address: Generative Programming and Component Engineering (GPCE),* Pittsburgh, Pennsylvania, October 2002, pp. 32-49.
36. Peri Tarr, Harold Ossher, William Harrison, and Stanley Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *International Conference on Software Engineering*, Los Angeles, California, May 1999, pp. 107-119.
37. Jos Warmer and Anneke Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.