A Model-Driven Approach to Enforce Crosscutting Assertion Checking

Jing Zhang, Jeff Gray and Yuehua Lin
University of Alabama at Birmingham
1300 University Boulevard
Birmingham, AL, 35294
205-934-5841
{zhangj, gray, liny} @ cis.uab.edu

ABSTRACT
Design by Contract provides an effective principle to enable the construction of robust software by describing properties of a module using logical assertions. This paper presents a model-driven approach for weaving assertion checking aspects into a large software system. The approach is based on a technique called two-level aspect weaving. At the top level, crosscutting assertions are weaved into a model by use of a model weaver. The second step of the weaving process occurs when the Model-Driven Program Transformation technique is applied to perform large-scale adaptation of the underlying source code from the contracts specified in the high-level models. The paper briefly presents a case study to illustrate the concept.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques, D.2.6 [**Software Engineering**]: Programming Environments – *graphical environments* and F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems.

Keywords
Design by Contract, Aspect-Oriented Programming, Model-Driven Software Development, Program Transformation.

1. INTRODUCTION
Design by Contract (DBC) is a well-known methodology [13] to help construct reliable and robust software. The basic idea of DBC is that a class can be viewed as having a contract with its client, whereby the client agrees to satisfy certain requirements before calling a method specified by the class (the pre-conditions of the class's method). Correspondingly, the class guarantees certain results after the execution of the method call (the post-conditions of its method).

ICSE MACS Workshop, May 2005, St. Louis, MO, USA.

Applying DBC can be a challenge. Manual placement of pre/post conditions into the application has serious drawbacks in terms of poor modularity and reusability. A desired solution to achieve modularization is that a change in a design decision is isolated to one location. Aspect-Oriented Programming (AOP) [9] has been investigated as an effective technique for improving modularization of crosscutting features. From the viewpoint of AOP, pre/post conditions that define systemic global properties represent a type of crosscutting concern [12], [5].

Most of the current DBC tools (e.g., JML [10] and Cona [16]) work primarily on the implementation's source code. However, software systems are not just source code, but rather collaborations of different representations of software artifacts (e.g., design models, configuration files, documents). The separations of concerns (e.g., the logical assertions in DBC) not only exists in the application code, but also are reflected at the design level. Thus, there is a need for tools that can employ DBC at different software abstraction levels to ensure the consistency between each level.

The main contribution of this paper is to provide a model-driven approach toward employing DBC at both the design model level and the implementation source level. The approach is based on a procedure called two-level aspect weaving. At the modeling level, the crosscutting assertions are weaved into a model by use of a model weaving tool; i.e., our tool called the Constraint-Specification Aspect Weaver (C-SAW) [6]. The second step of the weaving process occurs when the Model-Driven Program Transformation (MDPT) [7] technique is applied to perform large scale adaptation of the underlying source code from the contracts specified in the high-level models.

The paper is structured as follows. Section 2 introduces the general idea of two-level aspect weaving. A specific example for adding assertion checking aspects into a large software system is given in Section 3. The conclusion offers summary comments and a discussion of future work.

2. TWO-LEVEL ASPECT WEAVING

2.1 Constraint-Specification Aspect Weaver
C-SAW is a model transformation engine implemented as a plug-in component for the Generic Modeling Environment (GME) [1]. The GME is a domain-specific modeling tool that provides meta-modeling [8] capabilities to configure instance models from meta-

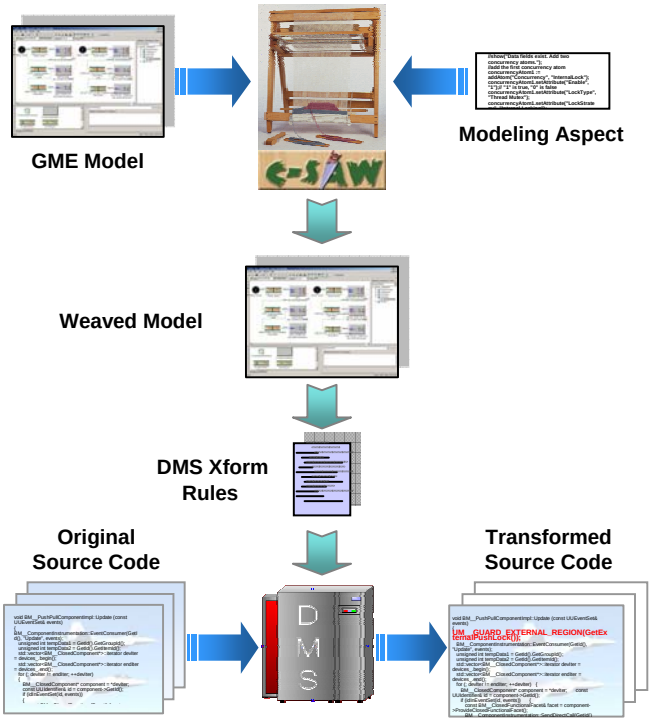

Figure 1. Two-level aspect weaving.

model specifications. C-SAW applies the ideas of AOP to GME to provide better modularization of model properties that are crosscutting throughout multiple layers of a model. C-SAW permits exploration of numerous modeling scenarios by considering crosscutting modeling concerns as aspects that can be inserted and removed from a model rapidly. This allows a modeler to make changes more easily to the base model without manually visiting multiple locations in the model.

The top of Figure 1 illustrates the model aspect weaving process in C-SAW where a base GME model serves as input to the model weaver (top-left of figure). This is the first level that performs weaving on models. C-SAW also requires a modeling aspect that captures the strategy for weaving a crosscutting modeling concern (top-right of figure). The output of C-SAW is a new model that has a crosscutting concern dispersed across the original base model. The modeling aspects are written in the Embedded Constraint Language (ECL) [6], which is an extension of OCL [17]. The ECL provides special operators to support model transformations within the GME.

ECL is distinct from OCL with respect to side-effects and model manipulation features. OCL is a declarative language and therefore it cannot support operations to create, update or remove the entities within a model, whereas the use of ECL requires the capability to introduce side-effects into a base model. This is needed because the modeling aspects specify transformations that must be performed on the model. This requires the ability to make modifications to the model. Therefore, ECL supports an imperative transformation procedural style with numerous operations that can alter the state of the model. An example of ECL will be presented later in Sections 3.

2.2 Model-Driven Program Transformation

A key challenge for model-driven development is the ability to maintain the fidelity between the mapping of model properties and the underlying source code. With respect to model-driven evolution, the majority of model-driven software development tools are well-equipped to generate and synthesize new software artifacts. However, support for parsing and invasively [2] transforming legacy application source code from higher-level models is not well-represented in the research literature. In order to address this limitation, this paper briefly demonstrates the feasibility of utilizing the power of a mature program transformation system to support large scale source-level transformations.

The bottom of Figure 1 corresponds to MDPT, a generative [4] approach that is applied to perform the adaptation of the source code of a software system from properties described in high-level models. This is the second level that performs weaving on source code. The approach synergistically extends the GME modeling process by incorporating the Design Maintenance System (DMS) [3] as the underlying program transformation engine. The core component of DMS is an Abstract Syntax Tree (AST) term rewriting engine that supports powerful capabilities for pattern matching and source transformation. DMS provides pre-constructed domains for several dozen languages. These domains are very mature and have been used to parse several million lines of code in various domains. Furthermore, an important feature of DMS is the source-to-source transformation rules that can be applied to modify a large cross-section of a code base.

In the MDPT approach, domain-specific model interpreters are constructed as GME plug-ins that are able to make the comparison between the old and new models, and then generate the DMS transformation rules from the evolving features described in a model. With the aid of these interpreters, developers do not have to handle the low-level DMS transformation rules. Therefore, the corresponding source code (bottom-left of figure), along with the generated transformation rules (middle of figure), serves as input to the DMS engine. As a result, the source programs will be modified and adapted to the new requirements that are reflected in the model changes. An essential characteristic of this model-driven process is the existence of a causal connection between the models and the underlying source representation. That is, as model changes are made to certain properties of a system, those changes must have a corresponding effect at the implementation level.

3. PRE/POST ASSERTION CHECKING

This section presents a case study for adding pre/post assertion checking into Boeing's Bold Stroke mission computing avionics system using the two-level aspect weaving approach. Within Bold Stroke, the primary software elements are a middleware run-time framework and thousands of application components that are implemented in several million lines of C++ code [15]. Specific components that comply with certain established architecture policies can be configured into various military aircraft flight scenarios.

Figure 2 shows the internal representation of a specific component with its data elements, facet/receptacle descriptors and other constituents of a model that are specifically intended to

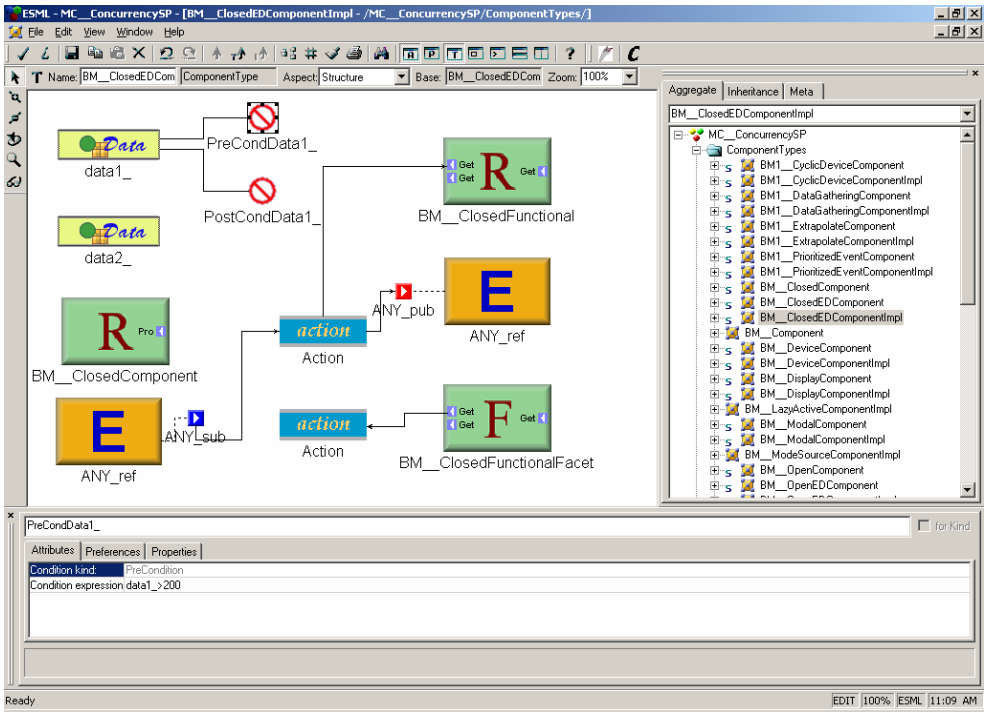

Figure 2. Internal representation of components in GME models.

describe the Bold Stroke component deployment and distribution middleware infrastructure. The infrastructure implements an event-driven model of computation. In this computation model, components update and transfer data to each other through event notification and call-back methods. Of particular interest to the context of this paper are the pre/post conditions attached to the data objects.

```
1  void BM__ClosedEDComponent::
2       Update(const UUEventSet& events)
3
4  {
5    assert(data1_>200);  // <- Precondition
6
7    BM_CompInstrumentation::
8          EventConsumer(GetId(), "Update", events);
9    unsigned int tempData1 = GetId().GetGroupId();
10   unsigned int tempData2 = GetId().GetItemId();
11
12   //* REMOVED code for Real-time Event Channel
13
14   data1_ = tempData1;
15   //* REMOVED actual variable names (proprietary)
16
17   data2_ = tempData2;
18
19   assert(data1<500); // <- Postcondition
20
21 }
```

Listing 1. C++ code fragment in "Update" method.

The equivalent C++ code fragment of the "`Update`" method from "`BM__ClosedEDComponent`" is shown in Listing 1. This method participates in the implementation of the real-time event channel and the update of the component data after performing all of the internal processing. Error conditions are anticipated either during system test or during actual system operation, such that the enforcement of DBC to the component implementations is desired.

In order to ensure the data validity throughout the whole control flow process, the pre/post conditions should be added into every component model of interest (see the "Pre/Post" modeling atoms in Figure 2). Furthermore, system developers need to insert the pre-condition assert statement (Listing 1, Line 5) at the entry of every "`Update`" method, and the post-condition (Listing1, Line19) at the end of the method in all of the corresponding components. Thus, any violation of the conditions that occurs during the execution is detected by these assertions.

An alternative to adding the contract assertions manually is to weave those assertions into the modeling level, and in turn drive the transformation of the corresponding code. To perform the model-level assertion weaving, Listing 2 shows the ECL code for weaving the pre/post conditions in Bold Stroke component models. The transformation strategy finds the "`data1_`" atom (Line 3 to Line 7) in every component whose name ends with "`Impl`" (line 28 to line 33). For each obtained "`data1_`" atom, two atoms are created, representing the pre-condition (Line 17-19) and post-condition (Line 22-24) with their condition expressions set to "`data1_>200`" and "`data1_<500`." Finally, these two conditions are connected to the "`data1_`" atom (Line 20 and Line 25). As a result, after using C-SAW to apply this ECL

specification, "PreCondData1_" and "PostCondData1_" atoms will be inserted into each component that has a "data1_" atom (see Figure 2).

```
1  defines Start, FindData1, AddConds;
2
3  strategy FindData1()
4  {
5    atoms()->select(a | a.kindOf() == "Data"
6        and  a.name() == "data1_")-> AddConds();
7  }
8
9  strategy AddConds()
10 {
11   declare p : model;
12   declare data, pre, post : atom;
13
14   data := self;
15   p := parent();
16
17   pre:=p.addAtom("Condition","PrecondData1_");
18   pre.setAttribute("Kind", "PreCondition");
19   pre.setAttribute("Expression", "data1_>200");
20   p.addConnection("AddCondition", pre, data);
21
22   post:=p.addAtom("Condition", "PostcondData1_");
23   post.setAttribute("Kind", "PostCondition");
24   post.setAttribute("Expression", "data1_<500");
25   p.addConnection("AddCondition", post, data);
26 }
27
28 aspect Start()
29 {
30   rootFolder().findFolder("ComponentTypes").
31   models()->select(m|m.name().endWith("Impl"))
32                                   ->FindData1();
33 }
```

Listing 2. ECL code for adding pre/post conditions in GME models.

```
1  default base domain Cpp~VisualCpp6.
2
3  pattern assertStmt() :
4          statement = "assert(data1_>200);".
5
6  pattern aspect(s:statement_seq):
7          statement_seq = " \assertStmt\(\){ \s }".
8
9  pattern joinpoint(id:identifier):
10         qualified_id = "\id :: Update".
11
12 rule precondition(ret:decl_specifier_seq,
13                   id:identifier,
14                   p:parameter_declaration_clause,
15                   s:statement_seq):
16     function_definition -> function_definition
17   =  "\ret \joinpoint \(\id\)(\p){\s}"
18   -> "\ret \joinpoint \(\id\)(\p){\aspect\(\s\)}"
19   if ~[modsList:statement_seq .s matches
20      "\:statement_seq \apect\(\modsList\)"].
21
22 public ruleset applyrules =
23 {
24   precondition
25 }.
```

Listing 3. Generated DMS rule to insert precondition statement into the C++ code.

After the pre/post conditions are weaved into the GME models, the next step is to invoke the MDPT interpreter to generate the DMS transformation rules. A fragment of the generated DMS transformation rule to insert the precondition statement into the Bold Stroke C++ code is presented in Listing 3. The DMS rule consists of declarations of patterns, rules, conditions, and rule sets using the external form (concrete syntax) defined by a language domain. The pattern in Line 3 represents the assert statement that is to be inserted. The second pattern (Line 6) describes the form of the resulting syntax tree. Pattern "joinpoint" (Line 9) provides the context in which the transformation rules will be applied. Here, the rules will be applied to all of the components containing an "Update" method. This pattern is similar to a join point in AspectJ [9]. The rule "precondition" (Line 12) represents a transformation of adding an assert statement at the beginning of each "Update" method. It is worth noting that the system developer does not create (or even see) the transformation rules. These are created by the MDPT interpreter and directly applied toward the transformation of Bold Stroke C++ source code using DMS. The resulting transformation will be equivalent to the DBC code that was manually added in Listing 1.

To summarize the whole process for implementing pre/post condition checking using two-level aspect weaving, the steps are:

- In the GME meta-model, include a new modeling element called "Condition." This addition is needed because the original Bold Stroke system does not support DBC;
- Associate different kinds of contracts to data fields in the component model using C-SAW;
- Make extension to the current MDPT interpreter according to the new requirements of the pre/post condition concern;
- Using the MDPT technique, weave specific "assert" statements into appropriate locations in the C++ source code according to different conditions specified in the corresponding GME model and ECL strategies.

4. CONCLUSION AND FUTURE WORK

This paper outlines a two-level aspect weaving approach to enforce DBC over different abstraction levels. An initial experiment was conducted to evaluate the feasibility of this approach using scenarios from an avionics application. Our model aspect weaver (C-SAW) has been successfully applied to many other different modeling languages [6], [19]. A model interpreter was developed to generate the program transformation rules needed to perform widespread source transformation of Boeing's Bold Stroke. In addition to the crosscutting concerns for DBC, the transformation process provides several other adaptations based on quality of service [14] policies specified in the models, such as concurrency control patterns and state management. We selected a subset of this system and applied adaptations across hundreds of C++ files that were successfully parsed and transformed in accordance with changes made in the representative models. For more technical details, see [7], [18].

Our initial investigation and associated prototype is tailored to a specific domain (i.e., real-time embedded avionics). The future work will focus on the generalization of the process for supporting software system evolution through two-level aspect weaving. In addition, we are in the process of developing a model testing suite to assist in assessing the correctness of model

transformations [11], such as those used to specify contracts at the modeling level. A debugging toolkit is also planned for C-SAW, which will be indispensable for detecting errors in the ECL specification during the weaving process.

The software, publications, and several video demonstrations related to this research can be obtained at http://www.cis.uab.edu/gray/Research/C-SAW.

5. ACKNOWLEDGMENTS

This work is supported by the DARPA Information Exploitation Office (DARPA/IXO), under the Program Composition for Embedded Systems (PCES) program.

6. REFERENCES

[1] *The Generic Modeling Environment: GME 4 User's Manual*, Institute for Software Integrated Systems, Vanderbilt University, 2004 (http://www.isis.vanderbilt.edu/Projects/gme/).

[2] Aßmann, U., *Invasive Software Composition*, Springer-Verlag, 2003.

[3] Baxter, I., Pidgeon, C., and Mehlich, M., "DMS: Program Transformation for Practical Scalable Software Evolution," *International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, May 2004, pp. 625-634.

[4] Czarnecki, K., and Eiseneker, U., *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[5] Diotalevi, F., "Contract Enforcement with AOP," *IBM DeveloperWorks*, July 2004, http://www-106.ibm.com/developerworks/library/j-ceaop/

[6] Gray, J., Sztipanovits, J., Schmidt, D., Bapty, T., Neema, S., and Gokhale, A., "Two-level Aspect Weaving to Support Evolution of Model-Driven Synthesis," in *Aspect-Oriented Software Development*, (Robert Filman, Tzilla Elrad, Mehmet Aksit, and Siobhán Clarke, eds.), Addison-Wesley, 2004, Chapter 30, pp. 681-710.

[7] Gray, J., Zhang, J., Lin, Y., Roychoudhury, S., Wu, H., Sudarsan, R., Gokhale, A., Neema, S., Shi, F., and Bapty, T., "Model-Driven Program Transformation of a Large Avionics Framework," *Generative Programming and Component Engineering (GPCE 2004)*, Springer-Verlag LNCS, Vancouver, BC, October 2004, pp. 361-378.

[8] Karsai, G., Maroti, M., Lédeczi, Á., Gray, J., and Sztipanovits, J., "Composition and Cloning in Modeling and Meta-Modeling," *IEEE Transactions on Control System Technology (special issue on Computer Automated Multi-Paradigm Modeling)*, March 2004, pp. 263-278.

[9] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W., "Getting Started with AspectJ," *Communications of the ACM*, October 2001, pp. 59-65.

[10] Leavens, G., and Cheon, Y., "Design by Contract with JML," Java Modeling Language Project, Internet: http://www.jmlspecs.org, 2003.

[11] Lin, Y., Zhang, J., and Gray, J., "A Testing Framework for Model Transformations," *Model-driven Software Development - Research and Practice in Software Engineering*, accepted for publication in 2005, a book by Springer.

[12] Lippert, M., and Lopes, C., "A Study on Exception Detection and Handling Using Aspect-Oriented Programming," *International Conference on Software Engineering (ICSE 2000)*, Limmerick, Ireland, 2000, pp. 418-427.

[13] Meyer, B., "Applying Design by Contract," *Computer*, 25(10), October 1992, pp. 40-51.

[14] Neema, S., Bapty, T., Gray, J. and Gokhale, A., "Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems," *First ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE '02)*, Springer-Verlag LNCS 2487, Pittsburgh, PA, October 2002, pp. 236-251.

[15] Sharp, D., "Component-Based Product Line Development of Avionics Software," *First Software Product Lines Conference (SPLC-1)*, Denver, Colorado, August 2000, pp. 353-369.

[16] Skotiniotis, T., and Lorenz, D., "Cona: aspects for contracts and contracts for aspects," *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, Vancouver, BC, Canada, October 2004, pp. 196-197.

[17] Warmer, J., and Kleppe, A., *The Object Constraint Language: Getting Your Models Ready for MDA*, Addison-Wesley, 2004.

[18] Zhang, J., and Gray, J., "Legacy System Evolution through Model-Driven Program Transformation," *EDOC Workshop on Model-Driven Evolution of Legacy Systems*, Monterey, CA, September 2004.

[19] Zhang, J., Lin, Y., and Gray, J., "Generic and Domain-Specific Model Refactoring using a Model Transformation Engine," *Model-driven Software Development - Research and Practice in Software Engineering*, accepted for publication in 2005, a book by Springer.