Model-Driven Program Transformation of a Large Avionics Framework

Jeff Gray[1], Jing Zhang[1], Yuehua Lin[1], Suman Roychoudhury[1], Hui Wu[1], Rajesh Sudarsan[1], Aniruddha Gokhale[2], Sandeep Neema[2], Feng Shi[2], and Ted Bapty[2]

[1] Dept. of Computer and Information Sciences, University of Alabama at Birmingham
Birmingham, AL 35294-1170
{gray,zhangj,liny,roychous,wuh,sudarsar}@cis.uab.edu
http://www.gray-area.org
[2] Institute for Software Integrated Systems, Vanderbilt University
Nashville, TN 37235
{gokhale,sandeep,fengshi,bapty}@isis.vanderbilt.edu
http://www.isis.vanderbilt.edu

Abstract. Model-driven approaches to software development, when coupled with a domain-specific visual language, assist in capturing the essence of a large system in a notation that is familiar to a domain expert. From a high-level domain-specific model, it is possible to describe concisely the configuration features that a system must possess, in addition to checking that the model preserves semantic properties of the domain. With respect to large legacy applications written in disparate programming languages, the primary problem of transformation is the difficulty of adapting the legacy source to match the evolving features specified in the corresponding model. This paper presents an approach for uniting model-driven development with a mature program transformation engine. The paper describes a technique for performing widespread adaptation of source code from transformation rules that are generated from a domain-specific modeling environment for a large avionics framework.

1 Introduction

A longstanding goal of software engineering is to construct software that is easily modified and extended. A desired result is to achieve modularization such that a change in a design decision is isolated to one location [24]. The proliferation of software in everyday life (e.g., embedded systems found in avionics, automobiles, and even mobile phones) has increased the level of responsibility placed on software applications [9, 29]. As demands for such software increase, future requirements will necessitate new strategies to support the requisite adaptations across different software artifacts (e.g., models, source code, test cases, documentation) [2].

Research into software restructuring techniques, and the resulting tools supporting the underlying science, has enhanced the ability to modify the structure and function of a software representation in order to address changing stakeholder requirements [16]. As shown in Figure 1, software restructuring techniques can be categorized as either horizontal or vertical. The research into horizontal transformation concerns modification of a software artifact at the same abstraction level. This is the typical connotation when one thinks of the term *transformation* [33], with examples being

G. Karsai and E. Visser (Eds.): GPCE 2004, LNCS 3286, pp. 361–378, 2004.

code refactoring [10] at the implementation level, and model transformation [4] and aspect weaving at a higher design level [13]. Horizontal transformation systems often lead to invasive composition of the software artifact [1]. In contrast, vertical transformation is typically more appropriately called *translation* (or synthesis) [33] because a new artifact is being synthesized from a description at a different abstraction level (e.g., model-driven software synthesis [12], [23], and reverse engineering). Vertical translations often are more generative in nature [8].

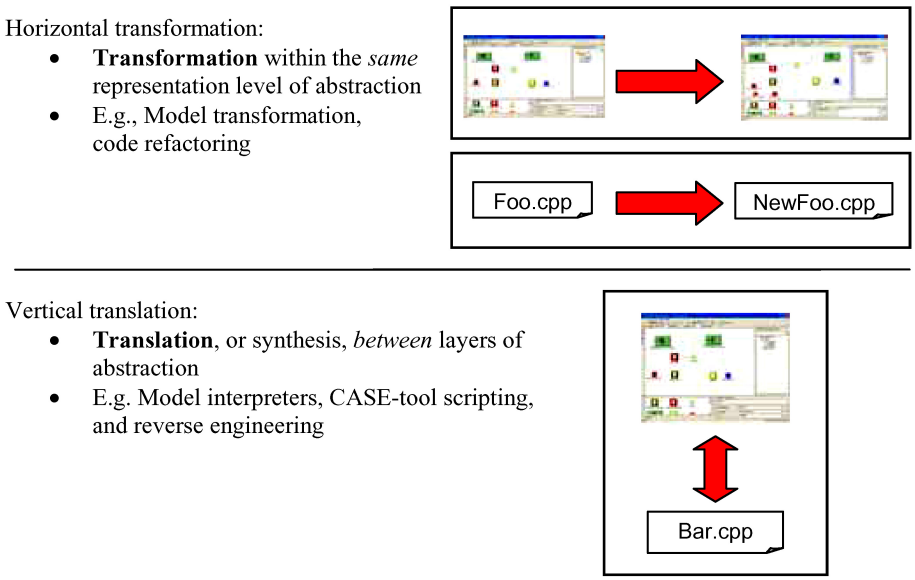

Fig. 1. Two directions of software transformation and translation

The most popular model-driven approach is the Object Management Group's (OMGs) Model-Driven Architecture (MDA), which separates application domain logic from the underlying execution platform [5, 11]. The overwhelming majority of early MDA efforts are of the translational, or synthesis style. That is, new software artifacts are generated whole-scale from properties that are refined from platform-independent models, down to platform-specific models, eventually leading to code. The challenge arises when MDA is to be applied to literally several hundred billion lines of legacy code in production use today [30]. To apply model-based techniques to such systems, it is beneficial to have an approach that is also transformational (i.e., one that actually modifies the source code representation) in order to add features to an existing code base. There are two primary factors that make it difficult to achieve true transformation of legacy source code from models:

- If pre-existing code is to be transformed from models, the model interpreters (existing within the modeling environment) must possess the ability to parse the underlying source. Thus, complex parsers must be built into the model interpreter. If a goal of the modeling environment is to achieve language-independence, then a new parser must be integrated into the model interpreter for each programming

language that is to be supported. This is very time consuming, if not unfeasible [20].

- Even if a mature parser is constructed for the underlying source, it is then necessary to provide a transformation engine to perform the adaptations to the source that are specified in the model. This is also a laborious task and was not needed by previous translators that only generated *new* artifacts from models. Yet, the need to synchronize model properties with pre-existing code requires the invasive capability for altering the code base.

We observe that the two difficulties enumerated above can be ameliorated by integrating the power of a program transformation system, which provides the required parsers and transformation engine, within a modeling tool chain. This paper describes our investigation into a synergistic technique that unites model-driven development (MDD) with a commercially available program transformation engine. Our approach enables adaptation of a large legacy system from properties described in a high-level model. A model interpreter generates the low-level transformation rules that are needed to provide a causal connection between the model description and the representative legacy source code. A video demonstration of the approach is available (please see Section 4 for details).

The rest of the paper is organized as follows: Section 2 provides an overview of the case study and the two technologies that are integrated to provide the realization of *Model-Driven Program Transformation (MDPT)*. A domain-specific visual modeling environment for embedded systems is introduced in Section 3. The heart of the approach is contained in Section 4. The fourth section also presents two illustrative examples of the approach applied to concurrency control and a black-box flight data recorder. The conclusion offers summary remarks, as well as related and future work.

2 Background: Supporting Technologies and Case Study

This paper unites the descriptive power provided by MDD (Section 2.1) with the invasive modification capabilities of a mature program transformation system (Section 2.2). Specifically, representative approaches from MDD and program transformation are described in this section to provide the necessary background to understand other parts of the paper. The mission computing framework that will be used as a case study is also introduced in Section 2.3.

2.1 Model-Integrated Computing

A specific form of MDD, called Model-Integrated Computing (MIC) [28], 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 automotive and avionics systems [29]) that tightly integrate the computational 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. The Generic Modeling Environment (GME) [21] is a meta-modeling tool based on MIC that can be configured and adapted from meta-level specifications (called the *modeling paradigm*) that describe

the domain [18]. When using the GME, a modeling paradigm is loaded into the tool to define an environment containing all the modeling elements and valid relationships that can be constructed in a specific domain. Thus, the approach provides a meta-environment for constructing system and software models using notations that are familiar to the modeler. The mission-computing avionics modeling environment described in Section 3 is implemented within the GME.

2.2 The Design Maintenance System

The Design Maintenance System (DMS) is a program transformation system and re-engineering toolkit developed by Semantic Designs (www.semdesigns.com). The core component of DMS is a term rewriting engine that provides powerful pattern matching and source translation capabilities [3]. In DMS parlance, a language domain represents all of the tools (e.g., lexer, parser, pretty printer) for performing translation within a specific programming language. DMS provides pre-constructed domains for several dozen languages. Moreover, these domains are very mature and have been used to parse several million lines of code, including the millions of lines of the targeted system explored in our research (i.e., Boeing's Bold Stroke [26], which is introduced in Section 2.3). Utilization of mature parsers that have been tested on industrial projects offers a solution to the two difficulties mentioned earlier; i.e., in addition to the available parsers, the underlying rewriting engine of DMS provides the machinery needed to perform invasive software transformations on legacy code [1]. Examples of DMS transformation rules will be given in Section 4.3.

2.3 The Bold Stroke Mission Computing Avionics Framework

Bold Stroke is a product-line architecture written in several million lines of C++ that was developed by Boeing in 1995 to support families of mission computing avionics applications for a variety of military aircraft [26]. As participant researchers in DARPA's Program Composition for Embedded Systems (PCES), we have access to the Bold Stroke source code as an experimental platform on which to conduct our research on MDPT. The following section describes the Bold Stroke concurrency mechanism that will be used later as an example to demonstrate the possibilities of our approach.

Bold Stroke Concurrency Mechanisms. To set the context for Sections 3 and 4, the Bold Stroke concurrency mechanism is presented to provide an example for the type of transformations that can be performed in order to improve better separation of concerns within components that have been specified in a domain-specific modeling language.

There are three kinds of locking strategies available in Bold Stroke: Internal Locking, External Locking and Synchronous Proxy. The Internal Locking strategy requires the component to lock itself when its data are modified. External Locking requires the user to acquire the component's lock prior to any access of the component. The Synchronous Proxy locking strategy involves the use of cached states to maintain state coherency through a chain of processing threads.

Figure 2 shows the code fragment in the "Update" method of the "BM_PushPull-Component" in Bold Stroke. This method participates in the implementation of a real-time event channel [17]. In this component, a macro statement (Line 3) is used to implement the External Locking strategy. When system control enters the Update method, a preprocessed guard class is instantiated and all external components that are trying to access the BM_PushPullComponent will be locked.

```
1 void BM__PushPullComponentImpl::Update (const UUEventSet& events)
2 {
3    UM__GUARD_EXTERNAL_REGION(GetExtPushLock());  // <-Locking Macro
4
5    BM_CompInstrumentation::EventConsumer(GetId(), "Update", events);
6    unsigned int tempData1 = GetId().GetGroupId();
7    unsigned int tempData2 = GetId().GetItemId();
8
9    //* REMOVED: code for implementing Real-time Event Channel
10
11   UM__GUARD_INTERNAL_REGION;  // <-Locking Macro
12   data1_ = tempData1;  //* REMOVED: actual var names (proprietary)
13   data2_ = tempData2;
14 }
```

Fig. 2. Update method in Bold Stroke BM_PushPullComponentImpl.cpp

After performing its internal processing, the component eventually comes to update its own data. At this point, another macro (Line 11) is used to implement the Internal Locking strategy, which forces the component to lock itself. Internal Locking is implemented by the Scoped Locking C++ idiom [25], which ensures that a lock is acquired when control enters a scope and released automatically when control leaves the scope. Specifically, a guard class is defined to acquire and release a particular type of lock in its constructor and destructor. There are three types of locks: Null Lock, Thread Mutex, and Recursive Thread Mutex. The constructor of the guard class stores a reference to the lock and then acquires the lock. The corresponding destructor uses the pointer stored by the constructor to release the lock.

The Problem with Macro-customization. The existence of locking macros, as shown in Figure 2, is representative of the original code base for Bold Stroke. During the development of that implementation, the concurrency control mechanisms implemented as locking macros occur in many different places in a majority of the components comprising Bold Stroke. In numerous configuration scenarios, the locking macros may evaluate to null locks, essentially making their existence in the code of no consequence. The presence of these locks (in lines 3 and 11 of Figure 2), and the initial effort needed to place them in the proper location, represents a point of concern regarding the manual effort needed for their initial insertion, and the future maintenance regarding this concern as new requirements for concurrency are added. The macro mechanism also represents a potential source of error for the implementation of new components – it is an additional design concern that must be remembered and added manually in the proper place for each component requiring concurrency control.

In Section 4, we advocate an approach that permits the removal of the locking macros (as well as other crosscutting properties) and offers automated assistance in adding them back into the code only in those places that are implied by properties

described in a model. Before describing that approach, however, it is essential to introduce the modeling language that is used to specify embedded systems like Bold Stroke.

3 Embedded Systems Modeling Language

In this section, the Embedded Systems Modeling Language (ESML) is described as a domain-specific graphical modeling language for modeling real-time mission computing embedded avionics applications. Its goal is to address the issues arising in system integration, validation, verification, and testing of embedded systems. ESML has been defined within the GME and is being used on several US-government funded research projects sponsored from DARPA. The ESML was primarily designed by the Vanderbilt DARPA MoBIES team, and can be downloaded from the project website at http://www.isis.vanderbilt.edu/Projects/mobies/. There are representative ESML models for all of the Bold Stroke usage scenarios that have been defined by Boeing.

3.1 ESML Modeling Capabilities

From the ESML meta-model (please see [21] for details of meta-model creation), the GME provides an instantiation of a new graphical modeling environment supporting the visual specification and editing of ESML models (see Figures 3 and 4). The model of computation used for ESML leverages elements from the CORBA Component Model [12] and the Bold Stroke architecture, which also uses a real-time event channel [17].

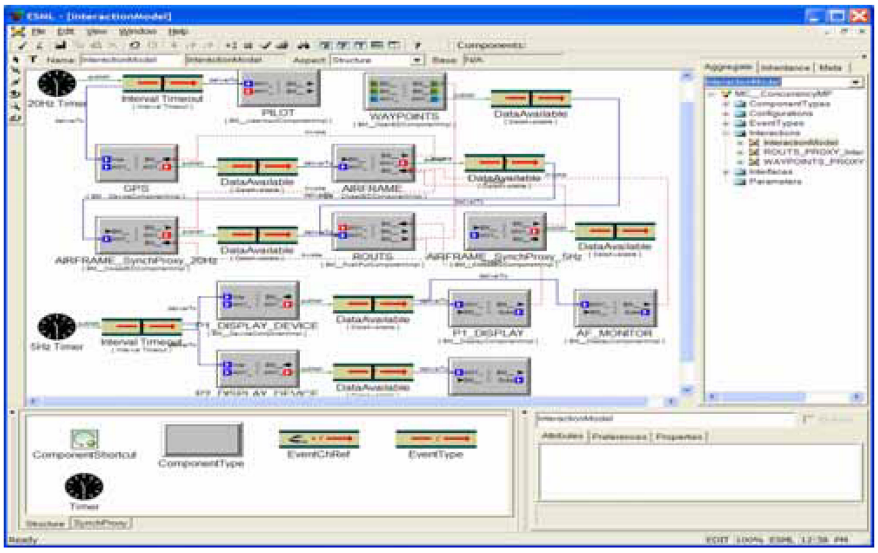

Fig. 3. Bold Stroke multi-threaded component interaction in ESML

The ESML provides the following modeling categories to allow representation of an embedded system: a) Components, b) Component Interactions, and c) Component Configurations. Figure 3 illustrates the components and interactions for a specific scenario within Bold Stroke (i.e., the MC_ConcurrencyMP scenario, which has components operating in a multi-processor avionics backplane). This higher-level diagram captures the interactions among components via an event channel. System timers and their frequencies are also specified in this diagram.

Figure 4 illustrates the ESML modeling capabilities for specifying the internal configuration of a component. The BM_PushPullComponent is shown in this figure. For this component, the concurrency control mechanism is specified, as well as facet descriptors, internal data elements, and a logging policy.

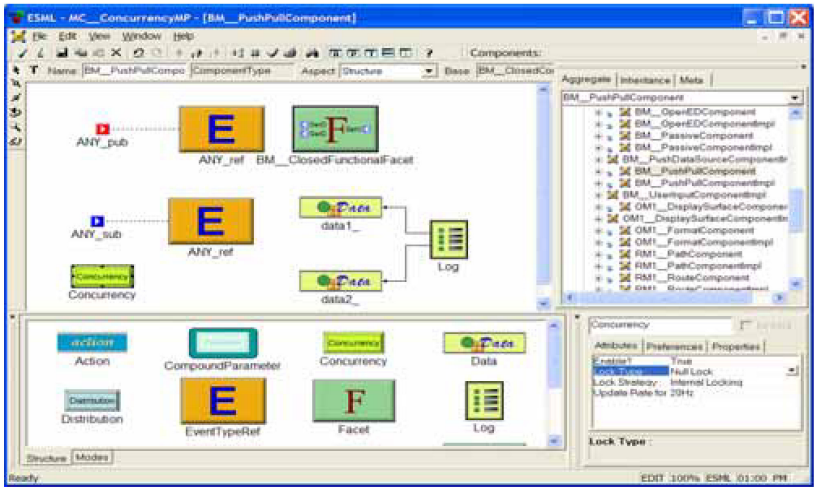

Fig. 4. Internal representation of the BM_PushPullComponent in ESML

3.2 ESML Model Interpreters

The result of modeling in ESML is a set of diagrams that visually depict components, interactions, and configurations, as shown in Figures 3 and 4. The objective of the design is to create, analyze, and integrate real systems; thus, we had to define a number of interfaces to support these activities.

A very important part of domain modeling within the GME is the capability of creating model interpreters. The modeling environment stores the model as objects in a database repository, and it provides an API for model traversal using a standard integration mechanism (i.e., COM) provided by the GME. Using the API, it is possible to create interpreters that traverse the internal representation of the model and generate new artifacts (e.g., XML configuration files, source code, or even hardware logic) based on the model properties. It is possible to associate multiple interpreters to the same domain.

Three model interpreters have been created for the ESML. The Configuration Interface interpreter is responsible for generating an XML file that is used during load-time configuration of Bold Stroke. The locking macros of Figure 2 are configured

from this generated file. The Configuration Interface provides an example of vertical translation that is more aligned with the synthesis idea for generating new artifacts, rather than a pure transformation approach that invasively modifies one artifact from descriptions in a model (as in Section 4). A second interpreter for ESML is the Analysis Interface, which assists in integrating third-party analysis tools. A third ESML interpreter has been created to invasively modify a very large code base from properties specified in an ESML model. This third interpreter enables the ideas of model-driven program transformation.

4 Model-Driven Program Transformation

The goal of model-driven program transformation (MDPT) is adaptation of the source code of a legacy system from properties described in high-level models. A key feature of the approach is the ability to accommodate unanticipated changes in a manner that does not require manual instrumentation of the actual source. An essential characteristic of the 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. A common way to achieve this correspondence is through load time configuration of property files that are generated from the models (e.g., the XML configuration file deployed by the Configuration Interface described in Section 3.2). There are two key problems with the load-time configuration file technique, however:

- The load time configuration mechanism must be built into the existing implementation. The source implementation must know how to interpret the configuration file and make the necessary adaptations at all of the potential extension points. For example, in Bold Stroke the locking strategy used for each component is specified in an XML configuration file, which is loaded at run-time during initial startup. The component developer must know about the extension points and how they interact with the configuration file at load time.
- A typical approach to support this load-time extension is macro tailorability, as seen in Figure 2. At each location in the source where variation may occur, a macro is added that can be configured from the properties specified in the XML configuration file. However, this forces the introduction of macro tags in multiple locations of the source that may not be affected under many configurations. The instrumentation of the source to include such tailoring is often performed by manual adaptation of the source (see lines 3 and 13 of Figure 2). This approach also requires the ability to anticipate future points of extension, which is not always possible for a system with millions of lines of code and changing requirements.

These problems provide a major hurdle to the transfer of model-based and load-time configuration approaches into large legacy systems. As an example, consider the two hundred billion lines of COBOL code that are estimated to exist in production systems [30]. To adopt the load-time configuration file approach to such systems will require large manual modifications to adjust to the new type of configuration. We advocate a different approach, based upon the unification of a program transformation system (DMS) with a modeling tool (GME).

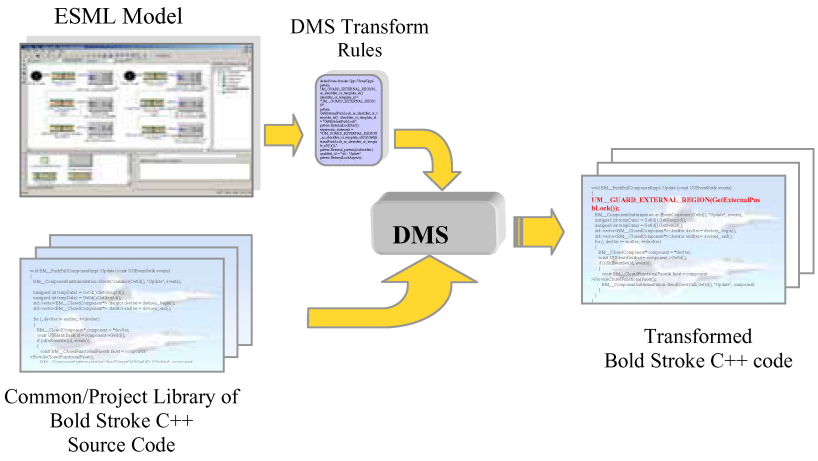

Fig. 5. Overview of Model-Driven Program Transformation

Figure 5 presents an overview of the idea of model-driven program transformation. The key to the approach is the construction of model interpreters that generate transformation rules from the model specifications. The rules are then fed into a program transformation system (represented in the top-left of Figure 5 that shows the path of generation from the models to the DMS transformation rules), along with the base implementation of a large application (e.g., Bold Stroke, as depicted in the bottom-left of the figure). The overall benefit of the approach is large-scale adaptation across multiple source files. The adaptation is accomplished through minimal changes to the models. Such super-linearity is at the heart of the abstraction power provided by model-driven techniques [14], [29].

In summary of Figure 5, the modeler simply makes changes to models using a higher-level modeling language, like the ESML. Those models are then interpreted to generate transformation rules that will invasively modify a large cross-section of an application. It should be noted that the modeler does not need to understand the accidental complexities of the transformation rule language. That process is transparent and is generated by the model interpreter. The following two sub-sections provide a description of crosscutting properties that have been weaved into the Bold Stroke C++ code from the model descriptions. The two examples represent crosscutting concerns related to concurrency control and recording of flight data information. A final subsection introduces the idea of two-level weaving, which allows aspects at the modeling level to drive widespread adaptations of the representative source code.

4.1 Weaving Concurrency into Bold Stroke

Recall the concurrency mechanism supported within Bold Stroke, as described in Section 2.3. In particular, consider the code fragment in Figure 2. There are a few problems with the macro tailorability approach, as used in this example code fragment:

- Whenever a new component is created, the developer must remember to explicitly add the macros in the correct place for all future components (a large source of error).
- Because a component may be used in several contexts, it is typical that different locking strategies are used in various usage scenarios. For example, the very existence of a Null Lock type is a direct consequence of the fact that a component is forced to process the macro even in those cases when locking may not be needed for a particular instantiation of the component. The result is that additional compile-time (or, even run-time overhead, if the chosen C++ compiler does not provide intelligent optimizations) is incurred to process the macro in unnecessary cases.

As an alternative, this paper presents a solution that does not require the locking to be explicitly added by the developer to all components. The approach only adds locking to those components that specify the need in a higher-level model, which is based on the requirements of the specific application scenario that is being modeled. This can be seen in the bottom-right of Figure 4, where the type of concurrency is specified for the selected "Concurrency" modeling atom (an internal null-lock is specified in this particular case). Suppose that all of the locking strategies did not exist in the component code (i.e., that lines 3 and 13 were removed from Figure 2 in ALL Bold Stroke components), and the component developers want to add the External Locking strategy to all of the hundreds of components that also require concurrency control. Completing such a task by hand is time-consuming and error-prone.

The DMS reengineering toolkit provides a powerful mechanism to transform code written in C++ and many other languages. In our investigation into the model driven program transformation approach, we initially removed the concurrency macros from a large set of components. We were able to insert different kinds of lock statements back into all of the Bold Stroke components that needed concurrency, as specified in the ESML models. This was accomplished by applying DMS transformation rules that were generated by a new ESML interpreter (see Section 4.3 for details).

4.2 Supporting a Black Box Data Recorder

In avionics systems, an essential diagnostic tool for failure analysis is a "black box" that records important flight information. This device can be recovered during a failure, and can reveal valuable information even in the event of a total system loss. There are several factors that make development of such a data recording device difficult:

- During ground testing and simulation of the complete aircraft system, it is often useful to have a liberal strategy for collecting data points. The information that is collected may come from a large group of events and invocations generated during testing of a specific configuration of Bold Stroke.
- However, an actual deployed system has very limited storage space to record data. In a deployed system, data may be collected from a small subset of the points that were logged during simulation. For example, only a few components may be of interest during specific phases of a mission. Also, only a subset of events may be recorded in an operational fighter jet.

It is a desirable feature to support the various types of recording policies that may be observed throughout development, testing, and deployment. Currently, the development tools associated with Bold Stroke do not support a capability to plug recording policies easily into the code base. The manual effort that would be required to plug/unplug different data recording policies throughout all components would be unfeasible in general practice. It is possible to transform existing Bold Stroke code by adding the black box flight recorder concern. The recorder information is specified by a logging policy (as can be seen in the "Log" modeling element of Figure 4). Within the logging policy, a modeler can specify policies such as "Record the values upon <entry/exit> of <a set of named methods>" or "Record the value upon every update to the <data variable>."

4.3 An Example of the Generated Transformation

The DMS Rule Specification Language (RSL) provides basic primitives for describing numerous transformations that are to be performed across the entire code base of an application. The RSL consists of declarations of patterns, rules, conditions, and rule sets using the external form (concrete syntax) defined by a language domain. Typically, a large collection of RSL files, like those represented in Figure 6 and Figure 7, are needed to describe the full set of transformations (we provide these two specifications as an illustration of the style of RSL that is generated from the ESML models). The patterns and rules can have associated conditions that describe restrictions on when a pattern legally matches a syntax tree, or when a rule is applicable on a syntax tree.

Figure 6 shows the RSL specification for performing two kinds of transformations: insertion of an External Locking Statement and an Internal Locking Statement. This RSL file was generated from the MDPT interpreter that we created, which extends the capabilities of ESML. The first line of the figure establishes the default language domain to which the DMS rules are applied (in this case, it is the implementation environment for Bold Stroke – Visual Studio C++ 6.0). Eight patterns are defined from line 3 to line 26, followed by two transformation rules. The patterns on lines 3, 6, 9, 13, 26 – along with the rule on line 28 – define the external locking transformation. Likewise, the patterns on lines 16, 19, 22 – and the rule on line 36 – specify the internal locking transformation.

Patterns describe the form of a syntax tree. They are used for matching purposes to find a syntax tree having a specified structure. Patterns are often used on the right-hand side (target) of a rule to describe the resulting syntax tree after the rule is applied. In the first pattern (line 3, Figure 6), a very simple pattern is described. This pattern matches the inserted macro (named **UM__GUARD_EXTERNAL_REGION**) to the syntax tree expression that is defined as **identifier_or_template_id** in the grammar definition of the DMS VC++6.0 domain. The third pattern (line 9) is used to combine the first and second pattern into a larger one, in order to represent the full macro statement along with its parameters. The target rule that describes the form of the resulting syntax tree is specified in the fourth pattern (line 13). This fourth pattern scopes the protected region and places the external locking statement as the first statement within the scope. Similarly, the pattern on line 22 describes the form of the resulting syntax tree after inserting an internal locking statement in front of any

```
1   default base domain Cpp~VisualCpp6.
2
3   pattern UM_GUARD_EXTERNAL_REGION_as_identifier_or_template_id():
4     identifier_or_template_id = "UM__GUARD_EXTERNAL_REGION".
5
6   pattern GetExternPushLock_as_identifier_or_template_id():
7     identifier_or_template_id = "GetExternalPushLock".
8
9   pattern ExternLockStmt(): expression_statement =
10    "\UM_GUARD_EXTERNAL_REGION_as_identifier_or_template_id\(\)
11    (\GetExternPushLock_as_identifier_or_template_id\(\)());".
12
13  pattern ExternLockAspect(s: statement_seq): compound_statement =
14                                    "{\ExternLockStmt\(\) {\s}}".
15
16  pattern InternLockStmt(): expression_statement =
17                                    "UM__GUARD_INTERNAL_REGION;".
18
19  pattern InternLockJoinPoint(expr:logical_or_expression): statement =
20                                    "data1_= \expr;".
21
22  pattern InternLockAspect(expr:logical_or_expression, s:statement_seq):
23    statement_seq = "\s {\InternLockStmt\(\)
24                         \InternLockJoinPoint\(\expr\)}".
25
26  pattern JoinPoint(id:identifier): qualified_id = "\id :: Update".
27
28  rule insert_extern_lock(id:identifier, s: statement_seq,
29                          p:parameter_declaration_clause):
30      function_definition -> function_definition =
31          "void \JoinPoint\(\id\)(\p) {\s} " ->
32          "void \JoinPoint\(\id\)(\p) {\ExternLockAspect\(\s\)}"
33  if ~[modsList:statement_seq. s matches
34             "\:statement_seq \ExternLockAspect\(\modsList\)"].
35
36  rule insert_intern_lock(expr:logical_or_expression, s:statement_seq):
37      statement_seq -> statement_seq =
38           "\s \InternLockJoinPoint\(\expr\)" ->
39          "\InternLockAspect\(\expr\,\s\)"
40  if s ~= "\:statement_seq \InternalLockStmt\(\)".
41
42  public ruleset applyrules={insert_extern_lock, insert_intern_lock}.
```

Fig. 6. A set of generated locking transformation patterns and rules in the DMS Rule Specification Language

update of data1_. The last pattern (line 26) 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 [19]. Although this last pattern is very simple, it quantifies over the entire code base and selects all of those syntax trees matching the pattern.

The RSL rules describe a directed pair of corresponding syntax trees. A rule is typically used as a rewrite specification that maps from a left-hand side (source) syntax tree expression to a right-hand side (target) syntax tree expression. As an example, the rule specified on line 28 of Figure 6 represents a transformation on all Update methods (specified by the JoinPoint pattern). The effect of this rule is to add an external locking statement to all Updates, regardless of the various parameters of each Update method. Notice that there is a condition associated with this rule (line 33). This condition describes a constraint that this rule should be applied only when there

already does not exist an external locking statement. That is, the transformation rule will be applied only once. Without this condition, the rules would be applied iteratively and fall into an infinite loop. The rule on line 36 applies the transformations associated with inserting an internal locking statement just before modification of the internal field named data1_. Rules can be combined into *sets* of rules that together form a transformation strategy by defining a collection of transformations that can be applied to a syntax tree. In the ruleset defined on line 42, the two locking rules are aggregated to perform a sequence of transformations (i.e., External/Internal Locking).

The logging transformation is much simpler and can be found in Figure 7. For this example, the "Log on Method Exit" logging policy is illustrated (this is specified as an attribute in the "Log" modeling element of Figure 4). The patterns on lines 3, 5, 8 – with the rule on line 10 – denote the update logging transformation. The pattern on line 5 shows the resulting form after inserting a log statement on all exits of the Update method. The corresponding rule on line 10 inserts the logging statement upon the exit of every Update method of every component.

It is important to reiterate that the modeler/developer does not create (or even see) the transformation rules. These are created by the ESML interpreter and directly applied toward the transformation of Bold Stroke code using DMS, as shown in Figure 5.

```
1  default base domain Cpp~VisualCpp6.
2
3  pattern LogStmt(): statement = "log.add(\"data1_=\" + data1_);".
4
5  pattern LogOnMethodExitAspect(s: statement_seq): statement_seq =
6                                      "\s \LogStmt\(\)".
7
8  pattern JoinPoint(id:identifier): qualified_id = "\id :: Update".
9
10  rule insert_log_on_method_exit(id:identifier, s:statement_seq,
11                                 p:parameter_declaration_clause):
12      function_definition -> function_definition =
13          "void \JoinPoint\(\id\)(\p) {\s} " ->
14          "void \JoinPoint\(\id\)(\p) {\LogOnMethodExitAspect\(\s\)}"
15  if ~[modsList:statement_seq. s matches
16          "\:statement_seq \LogOnMethodExitAspect\(\modsList\)"].
17
18  public ruleset applyrules={insert_log_on_method_exit}.
```

Fig. 7. A set of generated logging transformation patterns and rules in the DMS Rule Specification Language

With respect to the generalization of the process for supporting new concerns (other than concurrency and logging strategies as indicated above) in the Bold Stroke application through the MDPT technique, the following two steps are involved:

- If the current ESML metamodel does not provide the paradigm to specify the new concern of interest, it has to be extended to include the new model concepts in order to support the new requirements.
- The MDPT interpreter itself also has to be updated to generate the corresponding DMS transformation rules for the new concerns.

4.4 Transformation at the Modeling Level

It is interesting to note that the specification of modeling concerns can also cut across a domain model [13], in the same way that aspects cut across code [19]. That is, the specification of concurrency and logging concerns in a model may require the modeler to visit multiple places in the model. This is undesirable because it forces the modeler to spend much time adapting model properties. We have previously worked on a model transformation engine called the Constraint-Specification Aspect Weaver (C-SAW), which allows high-level requirements to be weaved into the model before the model interpreter is invoked [14].

The C-SAW transformation engine unites the ideas of aspect-oriented software development (AOSD) [19] with MIC to provide better modularization of model properties that are crosscutting throughout multiple layers of a model. Within the C-SAW infrastructure, the language used to specify model transformation rules and strategies is the Embedded Constraint Language (ECL), which is an extension of Object Constraint Language (OCL). ECL provides many common features of the OCL, such as arithmetic operators, logical operators, and numerous operators on collections. It also provides special operators to support model aggregates, connections and transformations that provide access to modeling concepts within the GME. There are two kinds of ECL specifications: an aspect, which is a starting point in a transformation process, describes the binding and parameterization of strategies to specific entities in a model; and a strategy is used to specify elements of computation and the application of specific properties to the model entities.

Utilizing C-SAW, a modeler can specify a property (e.g., "Record All updates to All variables in All components matching condition X") from a single specification and have it weaved into hundreds of locations in a model. This permits plugging/unplugging of specific properties into the model, enabling the generation of DMS rules resulting in code transformations. We call this process *two-level weaving* [14].

As an example, Figure 8 contains the ECL specification to connect "Log" atoms (of type "On Method Exit") to "Data" atoms in ESML models (see Figure 4). The transformation specification finds all of the "Data" atoms (line 3 to line 6) in every component whose name ends with "Impl" (line 21 to line 25). For each "Data" atom, a new "Log" atom is created, which has its "MethodList" attribute as "Update" (line 17). Finally, it connects this new "Log" atom to its corresponding "Data" atom (line 18). As a result, after using C-SAW to apply this ECL specification, "LogOnMethodExit" atoms will be inserted into each component that has a "Data" atom. As a front-end design capability, model weaving drives the automatic generation of the DMS rules in Figure 7 to transform the underlying Bold Stroke C++ source program.

Video Demonstration. The web site for this research project provides the software download for the model transformation engine described in Section 4.4 Additionally, several video demonstrations are available in various formats of the Bold Stroke transformation case study presented in this paper. The software and video demonstrations can be obtained at http://www.gray-area.org/Research/C-SAW.

```
1  defines Start, logDataAtoms, AddLog;
2
3  strategy logDataAtoms()
4  {
5    atoms()->select(a | a.kindOf() == "Data")->AddLog();
6  }
7
8  strategy AddLog()
9  {
10   declare parentModel : model;
11   declare dataAtom, logAtom : atom;
12
13   dataAtom := self;
14   parentModel := parent();
15   logAtom := parentModel.addAtom("Log", "LogOnMethodExit");
16   logAtom.setAttribute("Kind", "On Method Exit");
17   logAtom.setAttribute("MethodList", "Update");
18   parentModel.addConnection("AddLog", logAtom, dataAtom);
19 }
20
21 aspect Start()
22 {
23   rootFolder().findFolder("ComponentTypes").models().
24               select(m|m.name().endWith("Impl"))->logDataAtoms();
25 }
```

Fig. 8. ECL code for adding "LogOnMethodExit" to "Data" in ESML models

5 Conclusion

A distinction is made in this paper between *translational* approaches that generate new software artifacts, and *transformational* techniques that modify existing legacy artifacts. The model-driven program transformation technique introduced in Section 4 offers a capability for performing wide-scale source transformation of large legacy systems from system properties described in high-level models.

The major difficulty encountered in this project centered on the initial learning curve for DMS. Much time was spent in understanding the capabilities that DMS provides. After passing the initial learning curve, we believe that DMS offers a powerful engine for providing the type of language-independent transformation that is required for large-scale adaptation using model-driven techniques.

Related Work – There are related investigations by other researchers that complement the model-driven program transformation (MDPT) approach described in this paper. The general goals of MDA [5, 11], and the specific implementation of MIC with GME [21, 28], are inline with the theme of our paper. The main difference is that most model-driven approaches synthesize new artifacts, but the approach advocated in this paper provides an invasive modification of legacy source code that was designed without anticipation of the new concerns defined in the models.

The properties described in the models are scattered across numerous locations in the underlying source. Hence, there is also a relation to the work on aspect-orientation [19], adaptive programming [22], and compile-time meta-object protocols [6]. The manner in which the MDPT approach transforms the legacy code has the same intent as an aspect weaver. Our early experimentation with OpenC++ [6] and AspectC++

[27], however, suggest that the parsers in these tools are not adequate to handle the complexities that exist in the million lines of C++ code in Bold Stroke. However, DMS was able to parse the Bold Stroke component source without any difficulty. As an aside, we have also used DMS to define an initial approach for constructing aspect weavers for legacy languages [15]. With respect to aspects and distributed computing, the DADO project has similar goals [34], but does not focus on modeling issues.

As an alternative to DMS, there are several other transformation systems that are available, such as ASF+SDF [31], TXL [7], and Stratego [32]. We chose DMS for this project due to our ongoing research collaboration with the vendor of DMS (Semantic Designs). From this collaboration, we were assured that DMS was capable of parsing the millions of lines of Bold Stroke code. We have not verified if this is possible with other transformation systems.

Future Work – With respect to future work, there are several other concerns that have been identified as targets for Bold Stroke transformation (e.g., exception handling, fault tolerance, and security). We will also explore the transformation of Bold Stroke to provide the provisioning to support adaptation based on Quality of Service policies. Our future work will focus on adding support to the ESML and the associated interpreter in order to address such concerns. In addition, the generalization of a process for supporting legacy system evolution through MDPT will be explored.

Acknowledgements

This project is supported by the DARPA Program Composition for Embedded Systems (PCES) program. We thank David Sharp, Wendy Roll, Dennis Noll, and Mark Schulte (all of Boeing) for their assistance in helping us with specific questions regarding Bold Stroke. Our gratitude also is extended to Ira Baxter for his help during our group's initiation to the capabilities of DMS.

References

1. Uwe Aßmann, *Invasive Software Composition*, Springer-Verlag, 2003.
2. Don Batory, Jacob Neal Sarvela, and Axel Rauschmeyer, "Scaling Step-Wise Refinement," *IEEE Transactions on Software Engineering*, June 2004, pp. 355-371.
3. Ira Baxter, Christopher Pidgeon, and Michael Mehlich, "DMS: Program Transformation for Practical Scalable Software Evolution," *International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, May 2004, pp. 625-634.
4. Jean Bézivin, "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.
5. Jean Bézivin, "MDA: From Hype to Hope, and Reality," *The 6th International Conference on the Unified Modeling Language*, San Francisco, California, Keynote talk, October 22, 2003. (http://www.sciences.univ-nantes.fr/info/perso/permanents/bezivin/UML.2003/UML.SF.JB.GT.ppt)
6. Shigeru Chiba, "A Metaobject Protocol for C++," *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Austin, Texas, October 1995, pp. 285-299.
7. James Cordy, Thomas Dean, Andrew Malton, and Kevin Schneider, "Source Transformation in Software Engineering using the TXL Transformation System," Special Issue on Source Code Analysis and Manipulation, *Journal of Information and Software Technology (*44, 13) October 2002, pp. 827-837.

8. Krzysztof Czarnecki and Ulrich Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.
9. Eric Evans, *Domain-Driven Design: Tackling Complexity at the Heart of Software*, Addison-Wesley, 2003.
10. Martin Fowler, *Refactoring: Improving the Design of Existing Programs*, Addison-Wesley, 1999.
11. David Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley and Sons, 2003.
12. Aniruddha Gokhale, Douglas Schmidt, Balachandran Natarajan, Jeff Gray, and Nanbor Wang, "Model-Driven Middleware," in *Middleware for Communications*, (Qusay Mahmoud, editor), John Wiley and Sons, 2004.
13. Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck, "Handling Crosscutting Constraints in Domain-Specific Modeling," *Communications of the ACM*, Oct. 2001, pp. 87-93.
14. Jeff Gray, Janos Sztipanovits, Douglas C. Schmidt, Ted Bapty, Sandeep Neema, and Aniruddha Gokhale, "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.
15. Jeff Gray and Suman Roychoudhury, "A Technique for Constructing Aspect Weavers Using a Program Transformation System," *International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, March 22-27, 2004, pp. 36-45.
16. William G. Griswold and David Notkin, "Automated Assistance for Program Restructuring," *Trans. on Software Engineering and Methodology*, July 1993, pp. 228-269.
17. Tim Harrison, 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)*, Atlanta, Georgia, October 1997, pp. 184-200.
18. Gábor Karsai, Miklos Maroti, Ákos Lédeczi, Jeff Gray, and Janos Sztipanovits, "Type Hierarchies and Composition in Modeling and Meta-Modeling Languages," *IEEE Trans. on Control System Technology* (special issue on *Computer Automated Multi-Paradigm Modeling*), March 2004, pp. 263-278.
19. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-Oriented Programming," *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241, Springer-Verlag, Jyväskylä, Finland, June 1997, pp. 220-242.
20. Ralf Lämmel and Chris Verhoef, "Cracking the 500 Language Problem," *IEEE Software*, November/December 2001, pp. 78-88.
21. 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.
22. Karl Lieberherr, Doug Orleans, and Johan Ovlinger, "Aspect-Oriented Programming with Adaptive Methods," *Communications of the ACM*, October 2001, pp. 39-41.
23. 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)*, LNCS 2487, Pittsburgh, Pennsylvania, October 2002, pp. 236-251.
24. David Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, December 1972, pp. 1053-1058.
25. Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, John Wiley and Sons, 2000.
26. David Sharp, "Component-Based Product Line Development of Avionics Software," *First Software Product Lines Conference (SPLC-1)*, Denver, Colorado, August 2000, pp. 353-369.

27. Olaf Spinczyk, Andreas Gal, Wolfgang Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to C++," *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002),* Sydney, Australia, February 2002, pp. 53-60.
28. Janos Sztipanovits and Gábor Karsai, "Model-Integrated Computing," *IEEE Computer*, April 1997, pp. 10-12.
29. Janos Sztipanovits, "Generative Programming for Embedded Systems," *Keynote Address: Generative Programming and Component Engineering (GPCE),* LNCS 2487, Pittsburgh, Pennsylvania, October 2002, pp. 32-49.
30. William Ulrich, *Legacy Systems: Transformation Strategies*, Prentice-Hall, 2002.
31. Mark van den Brand, Jan Heering, Paul Klint, and Pieter Olivier, "Compiling Rewrite Systems: The ASF+SDF Compiler," *ACM Transactions on Programming Languages and Systems*, July 2002, pp. 334-368.
32. Eelco Visser, "Stratego: A Language for Program Transformation Based on Rewriting Strategies. System Description of Stratego 0.5," *12th International Conference on Rewriting Techniques and Applications (RTA),* Springer-Verlag LNCS 2051, Utrecht, The Netherlands, May 2001, pp. 357-361.
33. Eelco Visser, "A Survey of Rewriting Strategies in Program Transformation Systems," *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01) - Electronic Notes in Theoretical Computer Science*, vol. 57, Utrecht, The Netherlands, May 2001. (http://www1.elsevier.com/gej-ng/31/29/23/93/27/33/57007.pdf)
34. Eric Wohlstadter, Stoney Jackson, and Premkumar T. Devanbu, "DADO: Enhancing Middleware to Support Crosscutting Features in Distributed, Heterogeneous Systems," *International Conference on Software Engineering*, Portland, Oregon, pp. 174-186.