JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2010**

Vol. 9, No. 1, January-February 2010

A Program Transformation Technique to Support AOP within C++ Templates

Suman Roychoudhury, TATA Research Devolopment and Design Center, India.
Jeff Gray, University of Alabama, USA.
Jing Zhang, Motorola Applied Research Center, USA.
Purushotham Bangalore, University of Alabama at Birmingham, USA.
Anthony Skjellum, University of Alabama at Birmingham, USA.

Abstract

Aspect-oriented programming (AOP) provides assistance in modularizing concerns that crosscut the boundaries of system decomposition. Aspects have the potential to interact with many different kinds of language constructs in order to modularize crosscutting concerns. Although several aspect languages have demonstrated advantages in applying aspects to traditional modularization boundaries (e.g., object-oriented hierarchies), additional language concepts such as parametric polymorphism can also benefit from aspects. Many popular programming languages support parametric polymorphism (e.g., C++ templates), but the combination of aspects and generics is a topic in need of further investigation. The paper enumerates the general challenges of uniting aspects with C++ templates. It also underlines the need for new language constructs to extend AOP support to C++ templates and provides an initial solution to realize this goal.

1 INTRODUCTION

Aspect-Oriented Software Development [5] has shown initial promise in assisting a developer in isolating points of variation in a program. This helps the program to evolve and adapt to new change requirements during the lifecycle of the program. Aspects are language constructs that cleanly separate concerns that crosscut the modularization boundaries of an implementation. In a fundamentally unique way, aspects permit a software developer to quantify, from a single location, the effect of a concern across a body of code, thus improving the separation of crosscutting concerns. A translator called a weaver is responsible for merging the separated aspects with the base code. Some of the commonly used terminologies' in aspect-oriented programming (AOP) are:

Join Points are specific points of execution in a program, such as a method call, a constructor call or an object initialization.

Pointcuts are a means to identify a set of join points in a program and are expressed through a predicate expression that indicates a quantification over the code base.

Advice specify actions that are performed when a specific join point is matched by a given pointcut expression.

Inter-type[1] declarations allows crosscutting concerns to modify the structure of modules by declaring members or parents of another class in a single aspect such that the code related to a concern can reside in the same place.

Aspects define a modularization of a crosscutting concern for which the implementation might otherwise be distributed across multiple classes; aspects are expressed in terms of pointcuts and advice.

Motivation

The majority of research in the area of aspect-oriented programming [8] has focused on application to languages that support inheritance and subtype polymorphism (e.g., Java). There is potential benefit for applying the AOP concepts to other forms of polymorphism, such as parametric polymorphism [3], as found in languages that offer templates or generics (e.g., C++, Ada and Java). As a specific application area, aspects have the capability to improve the modularization of crosscutting concerns in large template libraries. Applying aspects to templates offers an additional degree of adaptation and configuration beyond that provided by parameterization alone.

The application of aspects to parametric polymorphism has not received much attention in the existing research literature. The most detailed discussion of aspects and C++ templates is described in [13], within the context of AspectC++ (an aspect language for C++) [17]. The effort to add aspects to templates in AspectC++ has been partitioned along two complimentary dimensions:

- Weaving advice into template bodies
- Using templates in the bodies of aspects

Whereas the AspectC++ work has focused along the second dimension (i.e., using templates in the aspect body), the key contribution of this paper is a deeper investigation along the first dimension (i.e., weaving advice in the template body).

In addition, the paper enumerates a key challenge pertaining to aspects and templates: Although a template is instantiated in multiple places, it may be the case that the crosscutting feature is required in only a subset of those instances (this challenge is further described in Section 2). For example, it may be required to weave in `vector` templates of type `int` only (i.e., `vector<int>`), leaving `vectors` of all other types

[1] The initial work presented in this paper does not consider inter-type declarations in its design.

unchanged. Additional language features are required to describe such specific intentions and is explained in detail in Section 2. Our solution is driven by a source-to-source pre-processor that utilizes a program transformation engine to perform the lower level adaptations. As a specific application area, we believe that there is a strong potential for impact if aspects can be used to improve the modularization of template libraries tailored for high-performance scientific computing. Such applications rely heavily on parametric polymorphism to specialize mathematical operations on vectors, arrays, and matrices [16], [18]. This paper presents the initial design and implementation of a template-aware aspect language.

Overview of Paper Contents

Section 2 presents an overview of the challenges and key concepts of AOP language design for C++ templates, and provides a solution technique in the design of a pointcut description language for templates. Section 3 shows the low-level implementation details that use a program transformation engine to perform the underlying weaving. Section 4 provides comparison to related work. A conclusion offers summary remarks and discusses future work.

2 AOP FOR C++ TEMPLATES

This section introduces several essential concepts of AOP for C++ templates. An application of the Standard Template Library (STL) [9] vector class is presented, along with a description of a program transformation technique for modularizing a crosscutting concern among vector instances. Initially, some of the elementary pointcut language constructs for C++ templates are introduced in Section 2. Later on, the paper motivates the need for advanced pointcuts for C++ templates and presents our initial approach to support this technique.

Simple Pointcut Expressions for C++ Templates

Listing 1 shows a simple implementation of class `Foo` that uses several instances of the STL vector class. The join point model and pointcut language are explained in terms of actual template definitions. The listing is purposely simplified so that the concepts are not complicated by peripheral details. There are three fields defined in `Foo`, either of type `vector<int>` or `vector<float>`. The methods `getMyInts` and `getMyFloats` return the corresponding vector field, and the method `addFloats` adds a new floating point number to a given floating point vector.

```
#include <vector>
using namespace std;

class Foo {
  public:
     vector<int>   getMyInts();
     vector<float> getMyFloats();
     void addMyFloats(vector<float>,float);
  protected:
     vector<int> myInts;
     vector<float>  myFloats;
     vector<float>  someOtherFloats;
  };

  vector<int> Foo::getMyInts() {
       return myInts;
  }
  vector<float> Foo::getMyFloats(){
       return myFloats;
  }
  void Foo::addMyFloats(vector<float> any,float aFloat) {
       any.push_back(aFloat);
  }
  ...
```

Listing 1: An example class with multiple template instantiations

Using `Foo` as a reference for discussion, some of the primitive pointcut expressions defined in our aspect language for C++ templates are explained below:

- A primitive `get` for the field `myFloats` is captured by the following pointcut expression:

`get(vector<*> Foo::myFloats)` **or**
`get(vector<float> Foo::myFloats)`

Note the wildcard "*" refers to any vector type.

- The *`execution`* of *all* "getters" (i.e., `getMyInts` and `getMyFloats`) is matched by the pointcut expression:

```
execution(vector<*> Foo::get*(..))
```

The expression "`get*`" matches all "get" methods.

- However, to match the `execution` of a *specific* get method (e.g., `getMyInts`), the above pointcut expression can be rewritten as:

```
execution(vector<int> Foo::getMyInts(..))
```

Here, instead of using a wildcard, we specify the exact method signature.

- Similarly, a `call` to the method `addMyFloats` is matched by the pointcut expression:

```
        call(void Foo::addMyFloats(vector<float>,float)).
```

In addition to the above examples, there are other pointcut expressions (e.g., set, constructor, initialization) that are available but not shown in this paper.

A key challenge that is addressed in our pointcut language design occurs from the realization that a template can be instantiated in multiple places, yet it may be the case that the crosscutting feature is required in only a subset of those instances. A generalized pointcut expression that quantifies over specific types may mistakenely capture several unintended instantiations. For example, if there are multiple `vector<float>` fields defined in class `foo`, it may be required to log a call only to the `push_back` method for the field `myFloats`, and leave other `vector<float>` fields (e.g., `someOtherFloats`) unaltered.

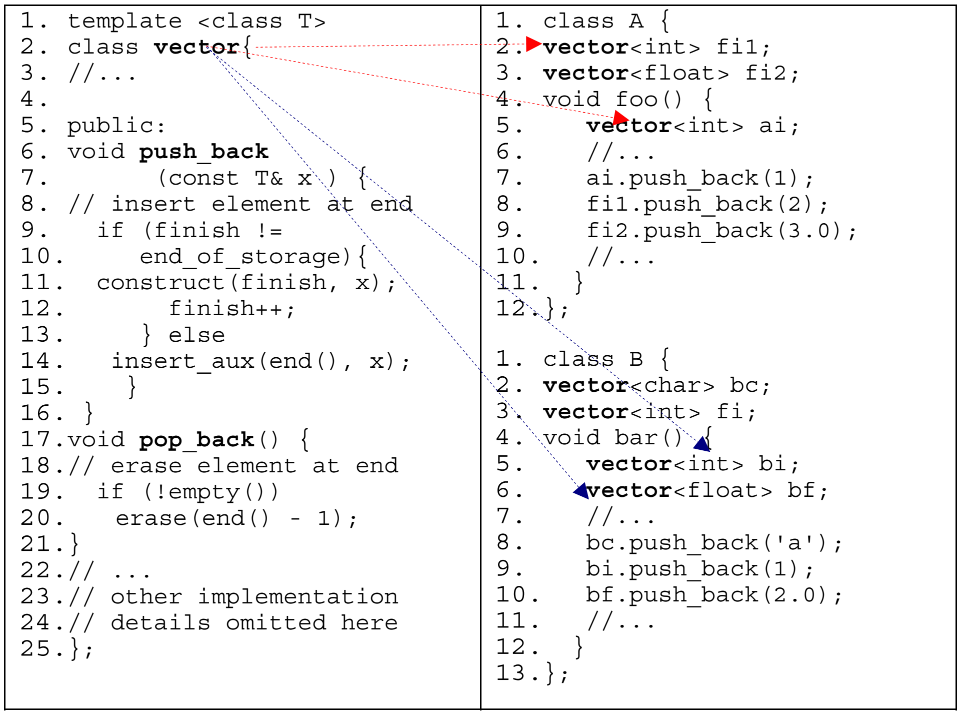
The flexibility to quantify over specific template instances provides additional power towards AOP in C++ templates that is not limited to specific types. However, a language mechanism is needed to define the quantification scope of a pointcut with respect to the semantics of C++ templates. The following section motivates the need for advanced pointcut expressions for C++ templates through a preliminary example.

Motivation for Advanced Pointcuts for C++ Templates

A fragment of the actual STL vector class definition is presented in Listing 2a, which shows the implementation of two vector-specific operations, `push_back` and `pop_back`. The sample code in Listing 2b illustrates the use of a vector in an application program. In this simple application, three different types of vector instances are declared (i.e., vectors of type `int`, `char`, and `float`). The `push_back` method is invoked on each vector instance to insert an element of a different type.

Considering the canonical logging[2] example, suppose that important data in specific vector instances needs to be recorded whenever the contents of the vector are changed. That is, within the context of an STL vector class, a requirement may state that logging is to occur for all items added to each execution of the `push_back` method, *but only for specific instantiations*. For example, it may be desired to log only vector fields of type `<int>` in class `A` (e.g., field `fi1` in class `A`) without affecting other local vector instantiations of type `int` in class `A` or `B` (e.g., those appearing in the local scope of method `foo` in class `A` or method `bar` in class `B`).

[2] We recognize the clichéd use of logging in AOP examples, but in this section we want to motivate the challenges of template weaving using a familiar concept.



```cpp
template <class T>
class vector{
//...

public:
void push_back
      (const T& x ) {
// insert element at end
  if (finish !=
     end_of_storage){
  construct(finish, x);
       finish++;
     } else
   insert_aux(end(), x);
    }
}
void pop_back() {
// erase element at end
  if (!empty())
   erase(end() - 1);
}
// ...
// other implementation
// details omitted here
};
```

```cpp
class A {
vector<int> fi1;
vector<float> fi2;
void foo() {
   vector<int> ai;
   //...
   ai.push_back(1);
   fi1.push_back(2);
   fi2.push_back(3.0);
   //...
  }
};

class B {
vector<char> bc;
vector<int> fi;
void bar() {
   vector<int> bi;
   vector<float> bf;
   //...
   bc.push_back('a');
   bi.push_back(1);
   bf.push_back(2.0);
   //...
  }
};
```

a) STL vector implementation b) Application using STL vectors

Listing 2: STL Vector Class and its usage

Challenges in Logging Specific Template Instance

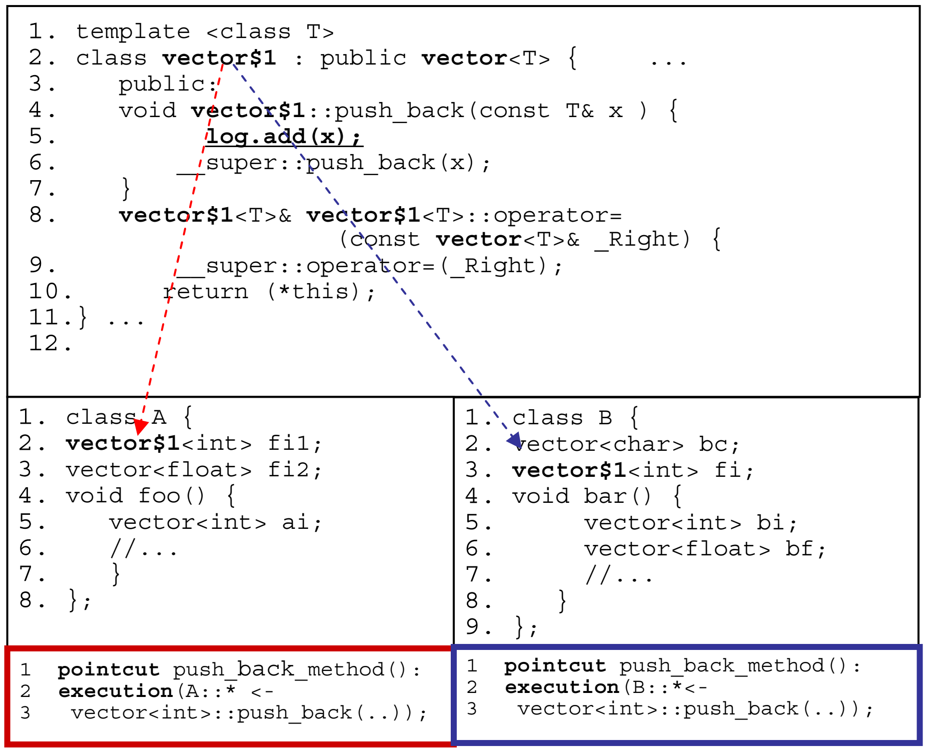
In order to record or log the contents of a given vector instance, the `push_back`[3] method as defined in the original vector template (Listing 2a) must be adapted. However, any change to this base template definition will affect all instantiations that reference the original vector template. For example, if logging support is added to the `push_back` method in the original vector template, all instantiations of vector (e.g., fields `fi1`, `fi2` in class `A`, fields `bc`, `fi` in class `B`, or method variables `bi` or `bf` in class `B`) will automatically implement support for logging. But according to the requirement, it is only desired to capture logging to specific instances of the vector (e.g., fields of type `vector <int>`) and not to all instances. The following section outlines an initial solution to address this challenge.

[3] The `push_back` method is invoked on each `vector` instance to insert an element of a different type.

Template Subtyping for AOP

In order to affect only `int` instances of the given vector template in fields of class `A` (or fields of class `B`) and leave other types (e.g., `float`, `char`) of vector instances unaltered, a new subtype `vector$1` is constructed, which inherits from the original vector template. The log statement is then added to the over-written `push_back` method of the `vector$1` template. The top-half of Listing 3 shows the adapted definition of the `push_back` method in this `vector$1` template. Note that the method call `log.add(x)` is added at the beginning of the `push_back` method in Listing 3. Finally, all field references in class `A` and `B` of type `vector<int>` are updated with this new `vector$1` template (shown in the middle of Listing 3). However, all other references to the original vector template are left unaltered (e.g., field `fi2` or method variable `ai` in class `A` retain their original declaration).

Although template specialization seems related to template subtyping, there could be instances where specialization may fail. For example, if only a particular instance of a specific type needs to be adapted (i.e., *only* the field `fi1` in class `A`), specialization techniques would fail as any specialization will be universally applied to *all* references of type `vector<int>` (e.g., method variable `ai` in class `A`). However, using template subtyping, *only the functions that need to be adapted are transformed with respect to the new aspect semantics*, but the rest of the class template remains unchanged.



```
template <class T>
class vector$1 : public vector<T> {     ...
   public:
   void vector$1::push_back(const T& x ) {
         log.add(x);
       __super::push_back(x);
   }
   vector$1<T>& vector$1<T>::operator=
                      (const vector<T>& _Right) {
         __super::operator=(_Right);
       return (*this);
} ...
```

```
class A {
vector$1<int> fi1;
vector<float> fi2;
void foo() {
    vector<int> ai;
    //...
    }
};
```

```
class B {
vector<char> bc;
vector$1<int> fi;
void bar() {
        vector<int> bi;
        vector<float> bf;
        //...
    }
};
```

```
pointcut push_back_method():
execution(A::* <-
 vector<int>::push_back(..));
```

```
pointcut push_back_method():
execution(B::*<-
 vector<int>::push_back(..));
```

Listing 3: STL `vector$1` class and updated references in the application instances

The following section describes in detail the scoping constructs required to specify the context of a given pointcut expression with respect to C++ templates.

Scoping Rules for Templates

This sub-section introduces the notion of advanced pointcut expressions based on the scope of a particular template instance. Table 1 illustrates the scoping rules for templates.

Scope Designator	Description
C::*	All global template instantiations of class C (fields)
* C.*(..)::*	All local template instantiations within all methods of class C
(C::* || * C.*(..)::*)	All template instantiations (both global and local) within class C
C.M(..)::*	All local template instantiations within method M of class C
* C.*(..)::V	Any template instantiation that is referenced by a variable V in all methods of class C
* C.M(..)::V	Template instantiation that is referenced by a variable V in method M of class C

Table 1. Scope designators in pointcut expressions

From the categorization of scope designators shown in Table 1, the example from Listing 3 can be re-visited to observe the scoping rules for classes `A` and `B` in the application program. At the bottom of Listing 3, two pointcut specifications are shown that capture the logging concern for specific `vector` instances depending on the scoping rule applied to the base class template. The pointcut in the bottom-left of Listing 3 can be read as, “select all fields of type `vector<int>` in *class* `A` that lead to an execution of the `push_back` method.” Similarly, the pointcut in the bottom-right of Listing 3 can be read as, “select all fields of type `vector<int>` in *class* `B` that lead to an execution of the `push_back` method.”

To illustrate this scoping rule further, additional examples are provided in Listings 4 through 8. Each pointcut definition is progressively more focused in limiting the scope of the join points that are captured (i.e., from a pointcut that captures all vectors of any type in any class, down to a pointcut that specifies a specific instance in a distinct method). Listing 4 offers an example of the aspect language to add the logging statement to the `push_back` method in *all* vectors of any type from *any* class. The pointcut `push_back_method` represents the points of `execution` where the advice is to be applied. In the pointcut expression, `vector<*>` denotes all types of vector instances.

```
template <class T>
aspect InsertPushBackLogToAllVector {
  pointcut push_back_method(const T& x):
     execution(vector<*>::push_back(..)) &&  args(x);
  before(const T& x):push_back_method(x) {
    log.add(x);
   }
}
```

Listing 4: Aspect specification for inserting the `push_back` log to all vectors of ANY type in ANY class

Listing 5 defines a pointcut that specifies the `execution` join point for the `push_back` method of all vectors of type `int`. The low-level implementation details involving the program transformation rules to automate the required changes to the template class and application program will be shown in Section 3.

```
pointcut push_back_method():
  execution(vector<int>::push_back(..));
```

Listing 5: Pointcut specification for weaving into all vectors of type `int` in ANY class

To add finer granularity, Listing 6 describes the pointcut specification for `execution` of all vectors of type `int` in class `A`. The operator '`<-`' is used to denote the scope of the vector template, however, a different symbol is chosen to distinguish from the standard C++ scope operator '`::`'. To be more specific in limiting the scope of a pointcut, Listing 7 defines a pointcut capturing all `int` vectors in method `foo` that are defined in class `A`.

```
pointcut push_back_method():
  execution((A::*  || * A.*(..)::*)<-
                vector<int>::push_back(..));
```

Listing 6: Pointcut specification for weaving into all vectors of type `int` in class `A`

```
pointcut push_back_method():
  execution(* A.foo(..)::*<-
                vector<int>::push_back(..));
```

Listing 7: Pointcut specification for weaving into all vectors of type `int` in method `foo` of class `A`

Listing 8 is the most specific pointcut expression; it will only match a particular template instance `ai` whose type is of `vector<int>` and is defined within the scope of method `foo` of class `A`.

```
pointcut push_back_method():
  execution(* A.foo(..))::ai<-
                 vector<int>::push_back(..));
```

Listing 8: Pointcut specification for weaving into vectors of type `int` and referenced by variable `ai` in method `foo` of class A

3 TEMPLATE WEAVING USING PROGRAM TRANSFORMATION

The aspect language shown in the previous section illustrates the high-level language specifically constructed to handle C++ templates. In this section, emphasis is placed on the low-level implementation details used to automate the weaving process through a program transformation engine.

The Design Maintenance System

The Design Maintenance System (DMS) [1] 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. In DMS terminology, 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.

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 (i.e., concrete syntax) defined by a language domain. 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 a transformation rule is applied. 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. 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. 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. Typically, a large collection of RSL files, like those represented in Listing 9 and Listing 10, are needed to describe the full set of transformations.

In addition to the RSL, a language called PARLANSE (PARallel LANguage for Symbolic Expressions) is available in DMS. Transformation functions can be written in PARLANSE to traverse and manipulate the parse tree at a finer level of granularity than provided by RSL. PARLANSE is a functional language for writing transformation rules as external patterns to provide deeper structural transformation. The DMS rules, along with the corresponding PARLANSE code, represent the transformations on the base STL library. However, due to the very low-level nature of the rewrite rules, it is not desirable that programmers be required to write their specifications using term rewriting or PARLANSE-specific functions. Instead, a high-level aspect language (similar to AspectJ, an aspect language for Java [10]) that hides the accidental complexities of RSL and PARLANSE from the programmer can be used to specify the weaving.

Figure 1 presents an overview of the automated transformation process that uses the DMS program transformation system as its underlying engine. One of the major components involved in the implementation of the weaver is the translator (bottom of figure), which parses and translates a high-level aspect language into low-level rewrite rules (i.e., referenced as items #5 and #6). This facilitates the application programmers to specify their intent using a high-level aspect language and remain oblivious to the existence of a low-level transformation engine.

The heart of the weaving process (core infrastructure) is the DMS transformation engine, which takes the source files and the generated rules as input. The user provides three different source files as input to the transformation process: the original STL source code (shown as item #1 in Figure 1), an application program based on the STL instances

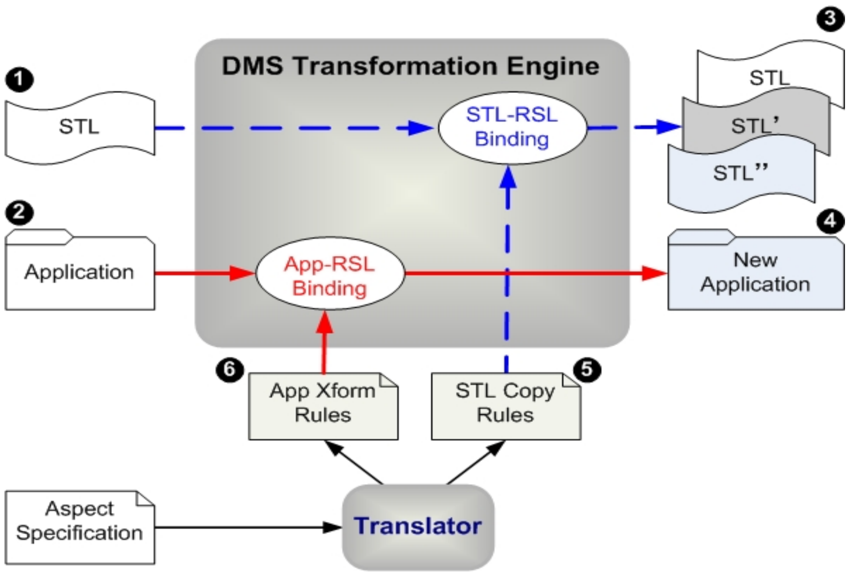


Figure 1: Overview of Template Weaving Process Applied to STL

(shown as item #2), and a high-level aspect language specification (examples shown in Section 2) used to describe the specific crosscutting concern with respect to template instantiations.

The translator includes a lexer, parser, and pattern evaluator (i.e., pattern parser and attribute evaluator) that takes the aspect specification and instantiates two different sets of parameterized transformation rules (i.e., STL copy rules and App transformation rules, shown separately as #5 and #6 in Figure 1). The pointcut expressions are bound to the corresponding transformation rules that are instantiated for matching patterns. The STL copy rules generate a subtype copy of the original STL class template by inheriting from the base template. The crosscutting concerns are weaved into this new subtype by overwriting appropriate methods as defined in the STL-RSL Binding. Note that each subtype copy rule encapsulates only one crosscutting concern for each specific template type (e.g., `vector<float>`). Therefore, it is desired to generate only one subtype copy for every type, each of which has one specific concern weaved into its base definition (shown as #3). However, if multiple concerns crosscut a specific type, then the corresponding subtype copy should also replicate this behavior by encapsulating multiple crosscutting concerns weaved into one copy. Similar to the STL-RSL Binding, the App-RSL Binding transformation modifies the user application program (shown as #2) based on the App transformation rules, and generates the new application (shown as #4) that is able to be compiled as a pre-processing phase and executed along with the generated subtype STL copies. The remainder of Section 3 provides a discussion of the transformation rules that implement these ideas.

Transformation Rules for Template Weaving

Listing 9 (STL template subtype copy rule, also shown as #5 in Figure 1) shows the low-level RSL specification for weaving a logging concern into the `push_back` method in an STL `vector` class. Two steps are involved in the weaving process: 1) make a subtype copy of the original vector template class, and 2) insert the logging statement into appropriate places in the abstract syntax tree. The first line of the rule establishes the default base language domain (e.g., C++) to which the transformations are applied.

```
default base domain Cpp.
pattern log_statement(): statement_seq = "log.add(x);".
pattern weaved_method_name(): identifier = "push_back".
pattern new_template_name(): identifier = "vector$1".
external pattern copy_template
   ( td : template_declaration,
     st : statement_seq,
     method_name : identifier,
  template_name : identifier ):
template_declaration = 'copy_template' in domain Cpp.

rule insert_log_to_template
( td : template_declaration ):
  template_declaration -> template_declaration
= td ->
  copy_template (td, log_statement(),
                    weaved_method_name(),
                    new_template_name()).

public ruleset applyrules = { insert_log_to_template }.
```

Listing 9: DMS transformation rules for weaving log statement into `push_back` method

Pattern log_statement in line 2 represents the log statement that will be inserted before the execution of the push_back method. Pattern weaved_method_name in line 3 defines the name of the method that will be transformed (i.e., push_back in this case). Pattern new_template_name in line 4 specifies the new name for the vector (i.e., vector$1).

As stated earlier, exit functions (i.e., external patterns and functions) in DMS are written in PARLANSE, which use internal APIs for performing various traversal and tree operations on the parsed abstract syntax tree. In this example, the external pattern copy_template (line 5 of Figure 9) is a PARLANSE function that performs the actual process of subtyping, naming, and weaving.

This external pattern takes four input parameters: 1) a template declaration to be operated on, 2) a statement sequence representing the advice, 3) a method name where the advice is to be weaved, and 4) a new name for the template subtype. The rule insert_log_to_template on line 12 triggers the transformation on the vector class by invoking the specified external pattern.

After applying this rule to the code fragment shown in Listing 2, a new template class named vector$1 (inherited from vector) will be generated with the logging statement inserted at the beginning of the push_back method (i.e., the automated result is the same as found in Listing 3). At this stage, the weaving process is still not complete because the application program also needs to be updated to reference the new vector$1 instance.

The DMS transformation rule to update the corresponding application program (App transformation rule, also shown as #6 in Figure 1) is specified in Listing 10. Pattern pointcut (lines 2 and 3) identifies the condition under which the rule will be applied (i.e., in this case, all int vector declarations). Pattern advice (lines 5 and 6) defines the name of the new transformed type (vector$1<int>). After applying this particular rule (line 21) to a given user application, the external pattern replace_vector_instance replaces the type of every template instantiation declared as type vector<int> into an instance of type vector$1<int>.

The notion of rewrite rules was introduced in this section not to perplex the interested reader, but to reveal the underlying details that enable the weaving mechanism to be applied to templates. Due to the low-level nature of the transformation rules, a programmer is not expected to write these rules. Rather, the aspect language mentioned in Section 2 and its corresponding binding with the RSL drives the weaving process. A programmer can specify the pointcut expression using the aspect language and the underlying rewrite rules are generated and correspondingly instantiated to match patterns in a manner that is transparent to the programmer.

```
default base domain Cpp.
pattern pointcut( id : identifier ):
     declaration_statement = "vector<int> \id;".

pattern advice( id : identifier ):
     declaration_statement = "vector$1<int> \id;".

external pattern replace_vector_instance
    ( cd  : class_declaration,
    ds1 : declaration_statement,
            ds2 : declaration_statement ):
 class_declaration = 'replace_vector_instance'
                     in domain Cpp.

rule replace_template_instance
 ( cd : class_declaration,
   id : identifier):
 class_declaration ->
 class_declaration
 = cd -> replace_vector_instance
        (cd,pointcut(id),advice(id)).
public ruleset applyrules = {replace_template_instance}.
```

Listing 10: DMS transformation rules to update
the application program

4 RELATED WORK

As noted in the introduction, a discussion of templates and aspects in AspectC++ within the context of generative programming is discussed in [13]. The focus of the AspectC++ work is on the interesting notion of incorporating parametric polymorphism into the bodies of advice. In contrast, the focus of our contribution is a deeper discussion of the complimentary idea of weaving crosscutting features into the implementation of template libraries. As an alternative to DMS, there are several other transformation systems that are available (e.g., ASF+SDF [2], TXL [4]) that could perhaps offer an alternative platform for the low-level transformation rules. With respect to the application of program transformation systems to aspect weaving, an investigation was described by Fradet and Südholt in an early position paper [6]. In similar work, Lämmel [11] discusses the implementation of an aspect weaver for a declarative language using functional meta-programs. Lopez et al. discussed the relationship of program transformation with respect to AspectJ [12]. In a different context [7], we applied program transformation technology to construct an aspect weaver for Object Pascal. This paper extends that work to C++ templates in order to address the challenges of transforming complex template code. AspectJ provides support for generics and parameterized types in pointcuts and intertype declarations. In order to restrict matching of patterns within given parameter types (for methods and constructors), return types (for methods) and field types, an appropriate parameterized type pattern is specified in the signature pattern of a pointcut expression.

Our initial work with C++ is also based on this idea. Additional flexibility is also provided by matching parameterized types within a given context using scope designators (as explained in Section 2).

5 CONCLUSIONS

Parametric polymorphism enables implementation of common algorithms and data structures in a type-independent manner. A template is contained in a single specification, but instantiated in multiple places within a target application. As shown in Section 2, applying aspects to templates raises several issues that need further investigation. For example, it is most likely that only a subset of the instances of a template is related to a specific crosscutting feature. In such cases, it would be incorrect to weave a concern naively into *all* template instantiations. A capability is needed to identify and specify those instances that are affected by an aspect, and to provide appropriate transformations that make a copy of the original template and weave on each copy. The study also illustrated the reason why adaptation has to be made not only to the template definition, but also to the application program that instantiates the template in multiple places.

Given the relevance of concern-based template adaptation, the contribution presented in this paper can be used for other programming languages that support parametric polymorphism (note: DMS provides mature grammars for several dozen languages). For instance, similar issues will arise with adoption of generics in other languages, as discussed by Silaghi [19].

Future directions will involve evaluating our weaver as applied to several scientific libraries that are implemented using C++ Templates (e.g., Blitz++ [18], POOMA [14], MTL [15]). An interesting topic that we will investigate is *library-independent* aspects that may exist within a specific domain, such as scientific computing, but applicable to several different libraries.

REFERENCES

[1] 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.

[2] 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.

[3] Luca Cardelli and Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, December 1985, pp. 471-522.

[4] James Cordy: "The TXL Source Transformation Language." *Science of Computer Programming*. 61(3): 190-210 (2006)

[5] Robert Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Aksit, *Aspect-Oriented Software Development*, Addison-Wesley, 2004.

[6] Pascal Fradet and Mario Südholt, "Towards a Generic Framework for Aspect-Oriented Programming," *Third AOP Workshop, ECOOP '98 Workshop Reader*, Springer-Verlag LNCS 1543, Brussels, Belgium, July 1998, pp. 394-397.

[7] 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 2004, pp. 36-45.

[8] John Irwin, Jean-Marc Loingtier, John Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman, "Aspect-oriented Programming of Sparse Matrix Code," *International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE)*, Springer-Verlag LNCS 1343, Marina del Ray, CA, December 1997, pp. 249-256.

[9] Nicolai M. Josuttis, The C++ Standard Library: A Tutorial and Reference, Addison-Wesley, 1999.

[10] 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.

[11] Ralf Lämmel, "Declarative Aspect-Oriented Programming," *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, San Antonio, TX, January 1999, pp. 131-146.

[12] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer, “A Disciplined Approach to Aspect Composition,” *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06)*, Charleston, SC, January 2006, pp. 68-77.

[13] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk, “Generic Advice: On the Combination of AOP with Generative Programming in AspectC++,” *Generative Programming and Component Engineering (GPCE)*, Springer-Verlag LNCS 3286, Vancouver, BC, October 2004, pp. 55-74.

[14] John V. W. Reynders, Paul J. Hinker, Julian C. Cummings, Susan R. Atlas, Subhankar Banerjee, William F. Humphrey, Steve R. Karmesin, Katarzyna Keahey, Marikani Srikant, and Mary Dell Tholburn, “POOMA: A Framework for Scientific Simulations of Paralllel Architectures,” in *Parallel Programming Using C++*, MIT Press, 1996.

[15] Jeremy Siek and Andrew Lumsdaine, “The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra,” *Computing in Object-Oriented Parallel Environments (ISCOPE)*, Springer-Verlag LNCS 1505, Santa Fe, NM, December 1998, pp. 59-70.

[16] Anthony Skjellum, Purushotham Bangalore, Jeff Gray, and Barrett Bryant, “Reinventing Explicit Parallel Programming for Improved Engineering of High Performance Computing Software,” *ICSE 2004 Workshop: International Workshop on Software Engineering for High Performance Computing System (HPCS) Applications*, Edinburgh, Scotland, May 2004.

[17] Olaf Spinczyk, Andreas Gal, and 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.

[18] Todd Veldhuizen, “Arrays in Blitz++,” *2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Springer-Verlag LNCS 1505, Santa Fe, NM, December 1998, pp. 223-230.

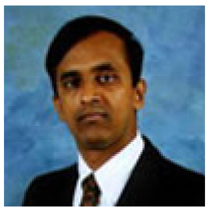
[19] Raul Silaghi and Alfred Strohmeier, “Better Generative Programming with Generic Aspects,” Second OOPSLA Workshop on Generative Techniques in the Context of MDA, Anaheim, CA, October 2003.

About the authors

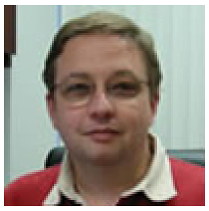
Suman Roychoudhury is a Research Scientist at the Tata Research Development and Design Center, Pune, India. He received his PhD from the University of Alabama at Birmingham (UAB). His research interests are energy-aware embedded systems, aspect-oriented software development, service-oriented computing and model-driven engineering. He can be reached at suman.roychoudhury@tcs.com.

Jeff Gray is an Associate Professor in the Department of Computer Science at the University of Alabama. His research interests include model-driven engineering, aspect-orientation, and generative programming. He can be reached at gray@cs.ua.edu.

Jing Zhang is a senior research engineer at Motorola Applied Research Center, where she is responsible for conducting research on adaptive and distributed architectures, event-driven system, and policy-based service orchestration. She received her PhD in Computer Science from the University of Alabama at Birmingham. She can be reached at j.zhang@motorola.com.



Purushotham Bangalore is an Associate Professor in the CIS Department at UAB. He has a Ph.D. in Computational Engineering from Mississippi State University. As Director of the Collaborative Computing Laboratory, Dr. Bangalore undertakes research in the area of Grid Computing Programming Environments. He can be reached at puri@cis.uab.edu.



Anthony Skjellum is Professor and Chair of the CIS Department at UAB. He received his Ph.D. in Chemical Engineering from the California Institute of Technology. He specializes in reusable, scalable mathematical software, and message passing middleware for scalable, real-time, and fault-tolerant systems. He can be reached at tony@cis.uab.edu.