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

Suman Roychoudhury, Jeff Gray, Jing Zhang, Purushotham Bangalore, and Anthony Skjellum

Dept. of Computer and Information Sciences, University of Alabama at Birmingham
Birmingham, Alabama, USA {roychous, gray, zhangj, puri, tony}@cis.uab.edu

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 with the emergence of generics in Java 5, the combination of aspects and generics is a topic in need of further investigation. The contribution of this chapter is a technique to support aspect-oriented programming within C++ templates using a program transformation approach. The chapter enumerates the general challenges of uniting aspects with templates through various examples. The core research idea presented in this chapter is applied to a large open-source C++ template library written for scientific computing.

1 Introduction

The majority of research in the area of aspect-oriented programming (AOP) 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 [4], as found in languages that offer templates or generics (e.g., C++ 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.

With the addition of generics¹ to Java [20], it can be expected that the application of aspects to parametric polymorphism will become a more focused research objective. However, the topic in general has not received much attention in the existing research literature. The most detailed discussion of aspects and templates is described in [18], within the context of the AspectC++ project [28]. The effort to add aspects to templates in AspectC++ has been partitioned along two complimentary dimensions:

¹ Throughout the remainder of this chapter, the term *template* is used to refer to the general concept of parametric polymorphism (even though there is a slight difference in meaning between the C++ term *template* and the *generic* term as used in Java).

1) weaving advice into template bodies, and 2) using templates in the bodies of aspects. The initial AspectC++ work is focused on the second dimension (templates in the advice body), but is limited by AspectC++'s difficulty in weaving into templates (please see related work for a discussion of this limitation). A contribution of this chapter is a deeper investigation of advice weaving in template implementation bodies.

The AspectJ 5² release provides support for generics and parameterized types in pointcut and inter-type declarations. However, a key challenge that is missing occurs from the realization that a generic is instantiated in multiple places, yet 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 2.2). Additional language features are needed to define the subset of template instantiations to which an aspect is to be applied, as well as an appropriate subtype copy semantics that is required to separate weaved templates from the base implementation. Furthermore, in addition to the transformation of the template itself, the application program that instantiates the template may also need to be altered according to the type of aspect applied to the template. This chapter discusses these issues in detail through several examples.

The technique described here is a source to source pre-processor that utilizes a program transformation system to perform the lower level adaptation that adds a crosscutting feature to a template implementation. In a different context [10], we applied program transformation technology to construct an aspect weaver for Object Pascal. This chapter extends that work to C++ templates in order to address the challenges of parsing and transforming complex templates.

Scientific computing was an initial application domain for the early examples of AOP [12]. However, aside from an application of AspectJ [15] to an implementation of JavaMPI [11], AOP has not been applied or investigated deeply within the area of scientific computing. This is primarily due to the fact that such applications are typically written in FORTRAN, C, or C++, but the center of AOP research has largely remained focused on Java-based implementations. Nevertheless, there is a strong potential for impact if aspects can be used to improve the modularization of template libraries tailored for parallel computation. Scientific computing applications written in C++ typically rely heavily on parametric polymorphism to specialize mathematical operations on vectors, arrays, and matrices [26, 30]. This chapter presents the initial design of a template-aware aspect language applied to a case study for a well-known scientific computing library.

The next section contains an overview of the challenges and key concepts of AOP language design for C++ templates, and provides a solution technique in the design of the pointcut description language for templates. Section 3 shows the program transformation details that provide a low-level virtual layering for the high-level aspect language. In Section 4, a popular open-source library for scientific computing serves as the case study for discussing crosscutting concerns that exist in template libraries. Section 5 provides comparison to other related work. A conclusion offers summary remarks and a vision for future work.

² Available at <http://www.eclipse.org/aspectj/> (AspectJ 5 is henceforth referred to as AspectJ).

2 AOP for C++ Templates

This section introduces a concise example that has been constructed to highlight several essential concepts of AOP for C++ templates. The example is purposely simplified so that the concepts are not complicated by peripheral details. An application of the Standard Template Library (STL) [14] vector class is presented, along with a description of a program transformation technique for modularizing a crosscutting concern among vector instances. STL is a general-purpose C++ library that embraces the idea of generic programming, which describes the implementation of algorithms and data structures (e.g., containers, iterators, algorithms, function objects and allocators) in a type-independent manner. Although the example in this section is simplified to improve the understandability of the approach, a case study representing the application of the idea to a popular scientific computing library is provided in Section 4.

2.1 An Introductory Example

This sub-section introduces several of the elementary AOP language constructs for C++ templates. The join point model and pointcut language are explained in terms of actual template definitions. Listing 1 shows the basic implementation of a class `Foo` that uses several instances of the STL `vector` class. There are numerous 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 `addmyFloats` adds a new floating point number to an existing `vector` field.

Using `Foo` as a reference for discussion, some of the primitive pointcut expressions for C++ templates can be explained. A primitive `get` for the field `myFloats` is matched by the pointcut expression `get(vector<*> Foo::myFloats)`, or by the pointcut `get(vector<float> Foo::myFloats)`. The execution of both methods `getmyInts` and `getmyFloats` can be matched by the pointcut expression `execution(vector<*> Foo:: get*(..))`; however, only `getmyInts` is matched by the expression `execution(vector<int> Foo::get*(..))`.

Similarly, a call to the method `addmyFloats` is matched by the pointcut expression `call(void Foo::add*(float))`. In addition to these examples, there are several other pointcut expressions that can be defined in an AspectJ-like style. However, a key challenge that is missing in AspectJ occurs from the realization that a template can be instantiated in multiple places, yet it may be the case that the *cross-cutting feature is required in only a subset of those instances*. A generalized pointcut expression (that quantifies over specific types only) may capture several unintended instantiations. For example, if there are multiple `vector<float>` fields defined in `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., `someOtherGuysFloats`) unaltered. Such flexibility to quantify over specific template instance variables (both fields and local variables) and not just limiting to specific types provides additional quantification power towards aspect-oriented programming in C++ tem-

plates. This challenge is further motivated in the next section. However, a language mechanism is needed to further define the quantification scope of a pointcut with respect to the semantics of C++ templates.

```
1. #include <vector>
2. using namespace std;
3. class Foo {
4.     public:
5.         vector<int> getmyInts();
6.         vector<float> getmyFloats();
7.         void addmyFloats(float aFloat);
8.     protected:
9.         vector<int> myInts;
10.        vector<float> myFloats;
11.        vector<float> someOtherGuysFloats;
12. };
13.     vector<int> Foo::getmyInts() {
14.         return myInts;
15.     }
16.     vector<float> Foo::getmyFloats() {
17.         return myFloats;
18.     }
19.     void Foo::addmyFloats(float aFloat) {
20.         myFloats.push_back(aFloat);
21.     }
```

Listing 1. An example class with multiple template instantiations

2.2 Template Subtype Copy Semantics

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 different type.

Considering the canonical logging example³ - suppose that important data in specific vector instances needs to be recorded whenever the contents 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 vectors of type `int` in method `foo` of class `A` (i.e., each time a `push_back` is executed on a `vector<int>` within the specific scope of class `A`) without affecting other `vector<int>` instantiations outside the scope of method `foo` in class `A`.

³ We recognize the clichéd use of logging in AOP examples, but in this introductory section we want to motivate the challenges of template weaving using a familiar concept.

<pre> 1. template <class T> 2. class vector{ 3. //... 4. 5. public: 6. void push_back 7. (const T& x) { 8. // insert element at end 9. if (finish != 10. end_of_storage){ 11. construct(finish, x); 12. finish++; 13. } else 14. insert_aux(end(), x); 15. } 16. } 17. void pop_back() { 18. // erase element at end 19. if (!empty()) 20. erase(end() - 1); 21. } 22. //... 23. // other implementation 24. // details omitted here 25. }; </pre>	<pre> 1. class A { 2. vector<int> ai; 3. void foo() { 4. vector<int> fi1; 5. vector<int> fi2; 6. vector<float> ff; 7. //... 8. ai.push_back(1); 9. fi1.push_back(2); 10. fi2.push_back(3); 11. ff.push_back(4.0); 12. //... 13. } 14. }; 1. class B { 2. vector<char> bc; 3. void bar() { 4. vector<int> bi; 5. vector<float> bf; 6. //... 7. bc.push_back('a'); 8. bi.push_back(1); 9. bf.push_back(2.0); 10. //... 11. } 12. }; </pre>
--	---

a) STL vector implementation

b) Application using STL vectors

Listing 2. STL Vector Class and its instances

In order to affect only `int` instances of the given `vector` template in method `foo` of class `A` and leave other types (e.g., `float`, `double`) of `vector` instances unaltered, a new subtype `vector_copy` is created, which is inherited from the original class template `vector` and then the `log` statement is added into the overwritten `push_back` method of the `vector_copy` class template. The copy of the `vector` class template fragment is shown in the top-half of Listing 3, which specifically adds the function call `log.add(x)` to the `push_back` method. As a consequence, all references in the application program to variables of type `vector<int>` in method `foo` of class `A` are updated with this new subtype, called `vector_copy<int>`.

In addition to the template class definition, the source code of the user application must also be updated to reference the new type `vector_copy` in appropriate places. In this case, all of the declaration statements of `vector<int>` in method `foo` of class `A` in the application program will now reference `vector_copy<int>` instead of the original `vector<int>`. The middle of Listing 3 shows the resulting changes that need to be made to the user application. Note that all other `vector` references (i.e., those that are instantiations of types other than `int`) are left unaltered.

<pre> 1. template <class T> 2. class vector_copy : public vector<T> { ... 3. public: 4. void vector_copy::push_back(const T& x) { 5. log.add(x); 6. super::push_back(x); 7. } 8. vector_copy<T>& vector_copy<T>::operator= (const vector<T>& _Right) { 9. super::operator=(_Right); 10. return (*this); 11. } ... 12. </pre>	
<pre> 1. class A { 2. vector<int> ai; 3. void fod() { 4. vector_copy<int> fi1; 5. vector_copy<int> fi2; 6. vector<float> ff; 7. //... 8. } 9. }; </pre>	<pre> 1. class B { 2. vector<char> bc; 3. void bar() { 4. vector_copy<int> bi; 5. vector<float> bf; 6. //... 7. } 8. }; </pre>
<pre> 1 pointcut push_back_method(): 2 execution(A.foo(..)::* <- 3 vector<int>::push_back(..); </pre>	<pre> 1 pointcut push_back_method(): 2 execution(B.bar(..)::bi <- 3 vector<int>::push_back(..); </pre>

Listing 3. STL vector_copy class and updated references in the application instances

Although partial or full template specialization may seem related to the subtype copy semantics, there could be further scoping rules where partial or full template specialization would not work. For example, if only a particular instance of a specific type needs to be adapted (i.e., only `vector<int> bi` in method `bar` of class `B`), specialization techniques would fail as any specialization will be universally applied to all references to type `vector<int>` and not just a particular one. To our knowledge, other aspect languages (e.g., AspectJ or AspectC++) do not handle scenarios that require a more selective application of the effect of advice applied to a template. Moreover, there are several advantages of subtype copy semantics, as summarized below.

As stated earlier, the idea presented in this chapter is not only to restrict our quantification over specific types, but also to quantify over types of specific variables. Section 2.2.1 will present a more detailed view of how specific template instances can be scoped based on the chosen pointcut expression. Now, returning to the advantages of

subtype copy semantics, let us re-visit Listing 1 presented in this chapter. In this example, two `vector<float>` instance variables are defined as follows:

```
vector<float> myFloats;  
vector<float> someOtherGuysFloats;
```

For a new requirement, if it is desired to add logging only to the `myFloats` instance variable, the technique allows to replace the type of `myFloats` to `vector_copy<float>` while the type of `someOtherGuysFloats` remains unchanged. However, a challenge that needs to be addressed is whether such an update will break any other parts of the program. For example, it is possible that there is an assignment like `myFloats = someOtherGuysFloats` in some other part of the program. This might result in type mismatch because subtyping ensures a relationship between the base and the derived class, which implies that a subtype can be assigned to its parent (e.g., `someOtherGuysFloats = myFloats;`), but the reverse is not true. Moreover, the assignment operator is not implicitly inherited as a result of subtyping, but the compiler will generate a default one if it is explicitly not present.

To avoid such scenarios, the top-half of Listing 3 shows an explicit definition of the overloaded `operator=` function for the derived class. It should be noted that the semantics of the assignment operation in the subtype (`vector_copy`) still remains the same as the original `vector`. That is, to maintain backward (or reverse) type compatibility it is only required to call the supertype's assignment operation and return the computed `this` pointer. This is a special case and needs to be handled explicitly to maintain backward type compatibility between sub and super types. All other binary and boolean operations are inherited by subtyping, which ensures its type safety. Thus, replacing the type of `myFloats` to `vector_copy<float>` would not cause any undue type mismatch, although it may be used subsequently in any arithmetic or boolean operation, or passed by reference to a member function, or returned as a function return type.

Another advantage of subtype copy semantics is that *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. The following piece of code investigates another challenge that may seem to represent a problem:

```
template<class T>  
class Bar{  
public:  
    vector<T> myTs;  
    vector<T> mySs;  
    bool foo(vector<T> myTs, vector<T> mySs){  
        if (myTs == mySs ) return true;  
        else return false;  
    }  
};
```

In the above code, if the type of `myTs` is changed to `vector_copy<float>` (to enable logging) in all instantiations on the form `Bar<float>`, there should not be any type error in method `foo`, because `vector_copy<float>` is a subtype of `vector<float>`, which ensures type safety between the two types.

The next sub-section describes in more detail a pointcut notation that provides additional scoping constructs to refine the context of a pointcut expression to name a subset of template instantiations for a specific type.

2.2.1 Scoping Rules for Templates

This sub-section discusses the design of a new pointcut expression notation that requires the creation of subtype copies of the original class template based on a refined scoping constraint. As an example, within application-specific instances of the `vector` class, there may be subtle points of variability required for a specific `vector` instance. To characterize this behavior, Table 1 illustrates the scoping rules for a pointcut description notation. The table is not a comprehensive list of all possible pointcut designators, but it captures the essential ones that will be explained below with suitable examples.

Pointcut Designator	Description
<code>C::*</code>	All template instantiations within the declaration of class <code>C</code> , which are not local instantiations in any methods of <code>C</code>
<code>* C.*(.)::*</code>	All local template instantiations within all methods of class <code>C</code>
<code>(C::* * C.*(.)::*)</code>	All template instantiations within class <code>C</code>
<code>C.M(..)::*</code>	All local template instantiations within method <code>M</code> of class <code>C</code>
<code>* C.*(.)::V</code>	Any template instantiation that is referenced by a variable <code>V</code> in all methods of class <code>C</code>
<code>* C.M(..)::V</code>	Template instantiation that is referenced by a variable <code>V</code> in method <code>M</code> of class <code>C</code>

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 variables of type `vector<int>` in method `foo` of class `A` that lead to an execution of the `push_back` method.” The pointcut in the bottom-right of Listing 3 can be read as, “select only variable `bi` of type `vector<int>` in method `Bar` of class `B` that leads to an execution of the `push_back` method.”

To illustrate this scoping issue 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.

```

1. aspect InsertPushBackLogToAllVector {
2.   pointcut push_back_method():
3.     execution(vector<*>::push_back(..));
4.
5.   before():push_back_method() {
6.     log.add(x);
7.   }
8. }

```

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.

```

1. pointcut push_back_method():
2.   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`. 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`.

```

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

```

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

```

1. pointcut push_back_method():
2.   execution(* A.Foo(..)) :*<-
3.     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 instance variable `fil` whose type is of `vector<int>` and is defined in method `Foo` of class `A`

```

1. pointcut push_back_method():
2.   execution(* A.Foo(..)) : fil<-
3.     vector<int>::push_back(..);

```

Listing 8. Pointcut specification for weaving into vectors of type `int` and referenced by variable `fil` in method `Foo` of class `A`

2.3 Template Copy Semantics for Partial and Full Template Specializations

In generic programming, one generic implementation of an algorithm is written for all possible data types such that each instance of the base template is implemented in exactly the same way. However, although most vectors might be implemented as arrays of the given type, vectors of a certain type T_1 may be implemented as vectors of certain type T_2 (e.g., `bools` might be represented as vectors of type `int` because most systems are going to use 16 or 32 bits for each boolean type even though it requires only a single bit; therefore, the boolean vector might be implemented differently where the data is represented as an array of integers and whose bits are manually manipulated). Listing 9 corresponds to vectors of all data types except that of type T_1 .

```

1. template <typename T>
2. class vector {
3.   private:
4.     T* vec_data; //store dynamically
5.     // other implementation details omitted
6. };

```

Listing 9. A simple vector template

To represent vectors of type T_1 , Listing 10 shows a full class template specialization of the given template.

```

1. template <>
2. class vector <T1> { // T1 is a concrete type
3.   private:
4.     unsigned T2 *vector_data; // T2 is a concrete type
5.     // other implementation details omitted
6. };

```

Listing 10. Full template specialization of the original vector template

However, additional scoping constraints may state that such type morphing is only applicable to certain instances of type T_1 in the application program with other instances remaining unaffected. The scoping expressions presented in Table 1 in conjunction with the template subtype copy semantics can address such restrictions. Listing 11 shows a pointcut expression for full template specialization applied to all vector instances of type T_1 in method `foo` of class `A`. That is, instances of type T_1 can use a subtype copy of the original vector template. Such controlled specialization techniques can give additional flexibility to enable AOP within C++ templates. The idea behind scoping template instances can be similarly applied to partial specialization techniques already available in C++.

```
1. pointcut morph_vector_of_type_T1() :  
2.   execution(void A.foo(..)::* <- vector<T1>;
```

Listing 11. Pointcut expression to scope vector instances of type T_1 in method `foo` of class `A`

3 Template Weaving using Program Transformations

The aspects shown in Section 2.2 illustrate the high-level aspect language for 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.

3.1 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 (surface 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 12 and Listing 13, 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 low-level 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) 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 process for implementing subtype template copy semantics. One of the major processes involved in the implementation is the translator (bottom of figure), which parses and translates a high-level aspect language into low-level transformation rules (i.e., referenced as item #5 and #6). As mentioned earlier, the application programmers are oblivious about the existence of a low-level infrastructure who can specify their intent using an AspectJ-like language. 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 process: the original STL source code (shown as item #1 in Figure 1), an application program based on the STL instances (shown as item #2), and an AspectJ-like language specification (examples shown in Section 2.2) that is 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 subtype 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 copy by overwriting appropriate methods as defined in the STL-RSL Binding. It also ensures type safety by providing an additional definition for the overloaded assignment operator, if required. Note that each subtype copy encapsulates only one crosscutting concern for each specific template type (`vector<float>`) or template instance variable (`vector<int> foo`). Therefore, it is desired to generate only one subtype copy for every type or instance variable, each of which has one specific concern weaved into its base definition (shown as #3). However, if multiple crosscutting concerns crosscut a particular instance variable or 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 and executed (as a pre-processing phase) along with the generated subtype STL copies. The remainder of Section 3 provides a discussion of the transformation rules that implement these ideas.

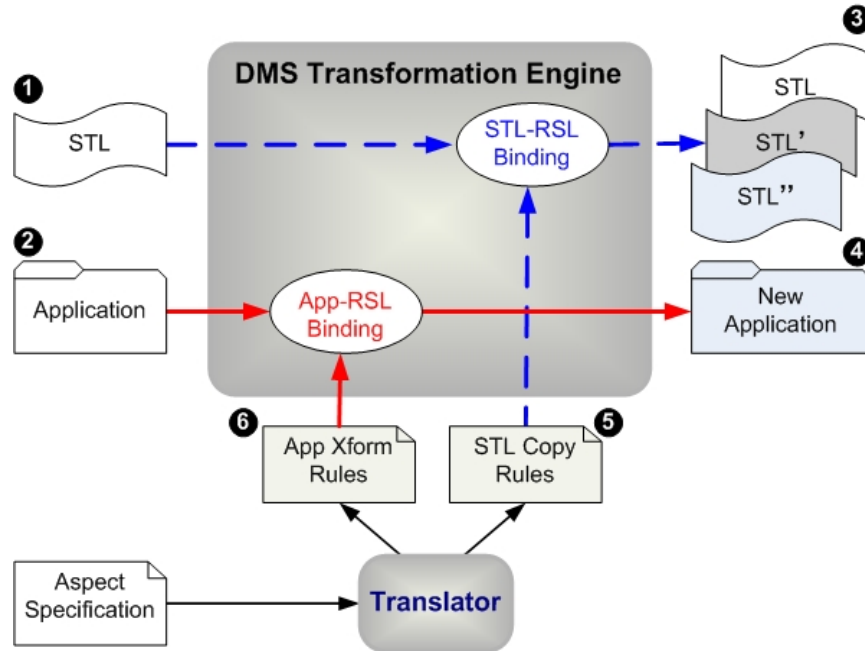


Fig. 1. Overview of Template Weaving Process Applied to STL

3.2 Transformation Rules for Template Weaving

Listing 12 (STL subtype copy rule, also shown as #5 in Fig. 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: make a subtype copy of the original vector template class, and insert the logging statement into appropriate places in the abstract syntax tree. The first line of the rule establishes the default base language domain to which the transformations are applied (in this case, Visual C++ 6.0 is used).

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 trans-

formed (i.e., `push_back` in this case). Pattern `new_template_name` in line 4 specifies the new name for the copied vector (e.g., `vector_copy`).

```

1.   default base domain Cpp~VisualCpp6.
2.   pattern log_statement(): statement_seq = "log.add(x);".
3.   pattern weaved_method_name(): identifier = "push_back".
4.   pattern new_template_name(): identifier = "vector_copy".
5.
6.   external pattern
7.     copy_template_add_log_to_pushback_method
8.     ( td : template_declaration,
9.       st : statement_seq,
10.      method_name : identifier,
11.      template_name : identifier ):
12.   template_declaration =
13. 'copy_template_add_log_to_pushback_method'
14. in domain Cpp~VisualCpp6.
15.
16.   rule insert_log_to_template
17. ( td : template_declaration ):
18.   template_declaration -> template_declaration
19. = td ->
20.   copy_template_add_log_to_pushback_method
21. ( td, log_statement(), weaved_method_name(),
22.   new_template_name() )
23. if td ~=
24.   copy_template_add_log_to_pushback_method
25. ( td, log_statement(), weaved_method_name(),
26.   new_template_name() ).

```

Listing 12. DMS transformation rules for weaving log statement into `push_back` method

As stated earlier, exit functions (i.e., external patterns and functions) in DMS are written in PARLANSE, which uses internal APIs for performing various traversal and tree operations on the parsed abstract syntax tree. In this example, the external pattern `copy_template_add_log_to_pushback_method` is a PARLANSE function that performs the actual process of subtyping, naming, and weaving. This external pattern takes four input parameters: a template declaration to be operated on, a statement sequence representing the advice, a method name where the advice is to be weaved, and a new name for the template subtype. Due to limited space, the PARLANSE code is omitted here; however, all of the transformation code used in this research, and related video demonstrations, are available on the GenAWeave project web page at <http://www.cis.uab.edu/roychous/genaweave>.

The rule `insert_log_to_template` on line 16 triggers the transformation on the vector class by invoking the specified external pattern. Notice that there is a condition associated with this rule (line 22), which describes a constraint stating that the rule should be applied only to those join points where a transformation has not occurred already. This is because the DMS re-write engine will continue to apply all

sets of rules until no rules can be fired. It is possible to have an infinite set of rewrites if the transformations are not monotonically decreasing (i.e., when one stage of transformation continuously introduces new trees that can also be the source of further pattern matches).

After applying this rule to the code fragment shown in Listing 2, a new template class named `vector_copy` (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 `vecor_copy` instance. The DMS transformation rule to update the corresponding application program (App transformation rule, also shown as #6 in Fig. 1) is specified in Listing 13. 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_copy<int>`). After applying this particular rule (line 26) to a given user application, the external pattern `replace_vector_instance` replaces the type of every instance variable declared as type `vector<int>` into an instance of type `vector_copy<int>`.

```

1.     default base domain Cpp~VisualCpp6.
2.     pattern pointcut( id : identifier ):
3.         declaration_statement = "vector<int> \id;".
4.
5.     pattern advice( id : identifier ):
6.         declaration_statement = "vector_copy<int> \id;".
7.
8.     external pattern replace_vector_instance
9.         ( cd : class_declaration,
10.          ds1 : declaration_statement,
11.          ds2 : declaration_statement ):
12.     class_declaration =
13.     'replace_vector_instance'
14.     in domain Cpp~VisualCpp6.
15.
16.     rule replace_template_instance
17.     ( cd : class_declaration,
18.       id : identifier):
19.     class_declaration ->
20.     class_declaration
21.     = cd -> replace_vector_instance
22.         (cd,pointcut(id),advice(id))
23.     if cd ~= replace_vector_instance
24.         (cd,pointcut(id),advice(id)).
25.
26. public ruleset applyrules = { replace_template_instance }.

```

Listing 13. DMS transformation rules to update the application program

The notion of low-level transformation rules were 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 rules in this manner. Rather, the aspect language mentioned in Section 2.2 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 low-level rules are generated and correspondingly instantiated (to match patterns) in a manner that is transparent to the programmer.

4 Case Study - Aspects in Scientific Libraries

This section focuses on modularizing open-source libraries written in the scientific computing domain. The contribution highlights the crosscutting features of a real case study and describes improved modularization using the concepts described in previous sections. In particular, this section focuses on aspects that we identified in Blitz++ [30], which is a C++ template library that supports high performance scientific computing.

4.1 Background: Crosscutting in Blitz++

Optimizing performance, while preserving the benefits of programming language abstractions is a major hurdle faced in scientific computing [21, 26, 29]. Object-oriented programming languages (OOPLs) have popularized useful features (e.g., inheritance and polymorphism) in the development of complex scientific problems. However, the performance bottleneck associated with OOPLs has been a major concern among high-performance computing (HPC) researchers. Alternatively, languages such as FORTRAN have dominated the numerical computing domain, even though the primitive programming constructs in such languages make applications difficult to maintain and evolve.

Compiler extensions (e.g., High Performance C++ [13] and High Performance Java [9]) and scientific libraries (e.g., POOMA [22], MTL [24], and Blitz++ [30]) have been developed to extend the benefits of object-oriented programming to the scientific domain. In particular, Blitz++ is a popular scientific package that has specific abstractions (e.g., arrays, matrices, and tensors) that support parametric polymorphism through C++ templates. The goal of the Blitz++ project has been to develop techniques that enable C++ to compete or exceed the speed of FORTRAN for numerical computing. Blitz++ arrays offer functionality and efficiency, but without any language extensions. The Blitz++ library is able to parse and analyze array expressions at compile time, and perform loop transformations. Blitz++ currently provides dense vectors and multidimensional arrays, in addition to matrices, random number generators, and tiny vectors. The overall size of the Blitz++ library is approximately 115K SLOCs. Moreover, there are several additional source code directories that serve as benchmarks and test cases.

Although Blitz++ makes extensive use of templates for array and vector implementation, the issue addressed in this section is the ability to apply AOP concepts to a large template library like Blitz++ using the technique described in Sections 2 and 3. This section contains a description of some of the array and vector implementation templates in Blitz++, and identifies several crosscutting features in the current Blitz++ implementation. The general approach could be applied to other languages that support parametric polymorphism (e.g., Ada and Java 1.5).

The first example (Section 4.2) represents the common case of a debugging precondition that appears 25 times in the `array-impl.h` source header, and in 57 places in `resize.cc`. A second crosscutting feature in `array-impl.h` is `setUpStorage`, which is used for initial memory allocation for arrays. `SetupStorage` appears 23 times in both `array-impl.h` and `resize.cc`. The third example (Section 4.3) is based on redundant assertion checks on the lower and upper bounds of an array during instantiation. It appears in 46 places in `array-impl.h`. This third example is similar in concept to the redundant assertions that were described by Lippert and Lopes in [17].

Section 4.4 examines AOP combined with other generative programming techniques [7]. In particular, the section explores the various binary and unary operations on vectors that use templated mathematical functions. These functions crosscut the vector operations. For example, many mathematical functions (e.g., `sin`, `cos`, `tan`, `abs`) are repeated in numerous places in both `vecuops.cc` and `vecbops.cc`. Although Blitz++ currently generates these templates, an alternative process is shown that uses transformation rules to generate the crosscutting mathematical functions. In the approach described in Section 4.4, over 12K SLOCs are generated using just 14 lines of code in a base template.

4.2 Precondition and SetupStorage Aspects

The Blitz++ library has a debugging mode that is enabled by defining the preprocessor directive `BZ_DEBUG`. In this mode, an application executes slowly because Blitz++ performs precondition and bounds checking on the array index. Under this condition, if an error or fault is detected by the system, the program halts and displays an error message. Listing 14 shows a sample precondition check for an array implementation. Note that the rank of the vector influences the precondition to be checked.

Another aspect that crosscuts the array implementation boundaries is `setUpStorage`. The method is called to allocate memory for any new array. However, any missing length arguments will have their value taken from the last argument in the parameter list. For example, `Array<int, 3> A(32, 64)` will create a 32x64x64 array, which is handled by the routine `setUpStorage`. Both the `BZPRECONDITION` (lines 10 and 20 of Listing 14) and `setUpStorage` (lines 12 and 22) can be individually considered as two different pieces of advice applied to the same pointcut (the former as before advice, and the latter as after advice).

```

1. template<typename T_expr>
2. _bz_explicit Array
3.     (_bz_ArrayExpr<T_expr> expr);
4.
5.     Array(int length0, int length1,
6.           GeneralArrayStorage<N_rank> storage =
7.           GeneralArrayStorage<N_rank>())
8.           : storage_(storage)
9.           {
10.            BZPRECONDITION(N_rank >= 2);
11.            // implementation code omitted
12.            setupStorage(1);
13.          }
14.
15.     Array(int length0, int length1, int length2,
16.           GeneralArrayStorage<N_rank> storage =
17.           GeneralArrayStorage<N_rank>())
18.           : storage_(storage)
19.           {
20.            BZPRECONDITION(N_rank >= 3);
21.            // implementation code omitted for brevity
22.            setupStorage(2);
23.          }
...

```

Listing 14. Precondition check and setupStorage in Blitz++ array implementation

With respect to the aspect language design presented in Section 2.2, Listing 15 contains a simple aspect specification for the crosscutting concern identified in Listing 14.

```

1. aspect InsertBZPreCondition_MemAllocation {
2.
3.   pointcut ArrayConstructor():
4.       execution(Array<*>::Array(..));
5.
6.   before() : ArrayConstructor() {
7.       BZPRECONDITION(N_rank >= tjp.getArgs().length());
8.   }
9.   after() : ArrayConstructor() {
10.      setupStorage(tjp.getArgs().length()-1);
11.   }

```

Listing 15. Aspect specification for precondition and memory allocation in templates

The expression statements in `BZPRECONDITION` and `setupStorage` form part of the before and after advice. The pointcut refers to all `Array` constructors defined in any `Array` type (denoted by the wildcard `*`). However, if it is desired to match

only arrays of type `Array<int>`, more selective pointcuts can be used. The function call `tjp.getArgs().length()` will return the length of the parameter list in the `Array` constructor. This is a special construct that is implemented internally by a `PARLANSE` external function.

4.3 Redundant Assertion Checking

Another debugging feature in `Blitz++` checks for the size or range of the arrays. To detect errors in ranges, each array allocation makes an implicit call to `assertInRange`, which checks the lower and upper bounds of the array. Listing 16 shows the internal implementation of the `assertInRange` function in `Blitz++`.

```
1. _bz_bool assertInRange(int BZ_DEBUG_PARAM(i0),
2.                       int BZ_DEBUG_PARAM(i1)) const
3. {
4.     BZPRECHECK(isInRange(i0,i1),
5.               "Array index out of range: ("
6.               << i0 << ", " << i1 << ") "
7.               << endl << "Lower bounds: "
8.               << storage_.base() << endl
9.               << "Length: " << length_ << endl);
10.    return _bz_true;
11. }
```

Listing 16. Definition of the assertion function

This particular assertion is defined in all array template specifications, according to the general pattern shown in Listing 17 (e.g., `assertInRange` in lines 4 and 9).

```
1. template<int N_rank2>
2.   T_numtype operator()
3.     (TinyVector<int,1> index) const {
4.     assertInRange(index[0]);
5.     return data_[index[0] * stride_[0]];
6.   }
7.   T_numtype operator()
8.     (TinyVector<int,2> index) const {
9.     assertInRange(index[0], index[1]);
10.    return data_[index[0] * stride_[0] +
11.                index[1] * stride_[1]];
12.   }
```

Listing 17. Redundant assertion check on base template specification

However, note that the number of index parameters passed to the `assertInRange` routine implicitly depends on the size of the `TinyVector`. For example, as presented in Listing 17, to allocate a `TinyVector` of size 1 requires a parameter (i.e., `index[0]`) to be passed to `assertInRange`. Similarly, for a vector of size 2, the range will be checked on `index[0]`, `index[1]`. This type of array specifi-

cation is repeated 46 times in `array-impl.h` and is context-dependent on the size of each template container.

To avoid the crosscutting assertion checking in every definition of an array implementation, the aspect specification (as defined in Listing 18) will weave this concern into the template code. The `operation_func` pointcut refers to all operation constructors in the array implementation class. The special construct `tjp.getParameterList` is internally mapped to a local PARLANSE function that returns part of the valid AST structure observed at this join point.

```
1. aspect AssertInRange {  
2.   pointcut operator_func():  
3.     execution(Array<*>::operator() (...));  
4.  
5.   before() : operator_func() {  
6.     assertInRange(tjp.getParameterList());  
7.   }
```

Listing 18. Aspect specification for redundant assertion checks

In our initial work, aspect mining and removal of the original crosscutting features was performed manually, although the actual weaving into the base code is automated with the low-level program transformation rules. Future work will explore aspect mining and clone detection within the context of templates, but is not a focus of this chapter.

4.4 Crosscutting Generic Functions

This section discusses the combination of AOP with other generative programming techniques [7]. In Blitz++, templates such as binary and unary operations for arrays and vectors are synthesized from a code generator implemented in several C++ routines. For consideration in this section, attention is focused on a specific set of unary vector (mathematical) operations in a template specification, which are generated to the `vecuops.cc` source file in the Blitz++ library containing around 12K SLOCs. Most of these mathematical operations (e.g., `log`, `sqrt`, `sin`, `floor`, `fmod`) have the same syntactic structure and can be specified concisely in the form of a pattern. An analysis of the generation process revealed that the entire template specification is essentially a cross-product between the set of defined mathematical operations (λ) and a base template (β) that represents the general pattern structure. In other words, the set of mathematical functions crosscut the entire unary vector general pattern.

If $\lambda_1, \lambda_2, \dots, \lambda_n$ represents the set of mathematical operations (e.g., `log`, `sin`, `sqrt`) that crosscuts the partial base template structure β (whole of Listing 19), then the code generated as the cross-product of λ and β can be represented as $\lambda_1\beta + \lambda_2\beta + \dots + \lambda_n\beta$. The partial string identifier **OPERATION** (highlighted in bold in Listing 19) identifies the locations in the partial base template structure where the mathematical operations are needed to be weaved to generate the whole template structure

(i.e., $\sum \lambda \times \beta = 12k$ SLOCs). The concept here is somewhat different than standard AOP practice, but the idea of a cross-product between a set of mathematical operations and a base pattern is germane to the overall process of template weaving. Although this example is based on vector operations using mathematical functions, similar situations (e.g., operations on Blitz++ arrays) exist in several other generated template specifications in the Blitz++ library.

```

1. template<class P_numtype1>
2. inline
3. _bz_VecExpr<_bz_VecExprUnaryOp<VectorIterConst
4.     <P_numtype1>,_bz_OPERATION<P_numtype1> >>
5.
6. OPERATION(const Vector<P_numtype1>& d1)
7.     {
8.         typedef _bz_VecExprUnaryOp<VectorIterConst
9.             <P_numtype1>,_bz_OPERATION<P_numtype1>> T_expr;
10.
11.         return _bz_VecExpr<T_expr>(T_expr(d1.begin()));
12.     }

```

Listing 19. Subset of base pattern used to generate the vector operation template

5 Related Work

As noted in the introduction, a discussion of templates and aspects in AspectC++ within the context of generative programming is discussed in [18]. 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. We were forced to adopt a program transformation approach because of the inability of current C++ aspect weavers to weave into the templates found in Blitz++ or STL. In fact, the current publically available version of AspectC++ (1.0pre2, distributed on December 21, 2005, and available at <http://www.aspectc.org/>) is *not able to weave into template definitions*. According to the most recent manual for AspectC++⁴, “Currently ac++ is able to parse a lot of the (really highly complicated) C++ templates, but weaving is restricted to non-templated code only. That means you can not weave in templates or even affect calls to template functions or members of template classes.”

We chose DMS for our experimentation because of our assurance of the ability to parse and transform all of the complex template specifications in STL and Blitz++. Many commercial C++ compilers do not implement enough of the ISO/ANSI C++ standard to compile all of Blitz++. As an alternative to DMS, there are several other transformation systems that are available (e.g., ASF+SDF [3], TXL [6]) 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

⁴ <http://www.aspectc.org/fileadmin/documentation/ac-compilerman.pdf>

investigation was described by Fradet and Südholt in an early position paper [8]. In similar work, Lämmel [16] discusses the implementation of an aspect weaver for a declarative language using functional meta-programs. A recent paper by Lopez et al. discusses the relationship of program transformation with respect to AspectJ [19].

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. In addition, AspectJ provides advanced features like declaring pointcuts inside the generic type definition and declaring generic aspects that allow an abstract aspect to be declared as a generic type. Our initial work on modularizing C++ templates does not provide enhanced features such as these and represents a future area of work. However, as stated earlier in this chapter, sometimes it may be required to adapt not only the specific types (return types, field types, parameter types), but also to adapt the specific instant variables that correspond to a specific type (especially in cases of C++ templates). To adapt to such stakeholder requirements, modification is needed to both the template definition and application specific template instances. Such a flexible feature to selectively quantify over template instantiation (both specific types and variables that refers to specific types) to enable AOP within C++ templates (or Java generics) is not supported by AspectC++ (or AspectJ). Moreover, certain restrictions are required to be applied by scoping the parameterized type patterns within valid boundaries in the given class structure. The core contribution of this chapter is driven by this notion and is realized from the principle of template subtype copying.

Within the scientific computing domain, ROSE provides optimizations using source to source transformation of ASTs for C++ applications [21]. The transformations are expressed using a domain-specific language [23]. The type of transformations performed by ROSE are focused solely on optimization issues of scientific libraries, and are not applicable to the kinds of transformations we advocate in this chapter to improve the modularization of crosscutting concerns in scientific code bases. In [5], Chalabine and Kessler have suggested seven different forms of interdependent concerns that are necessary to introduce parallelism within sequential programs.

6 Conclusion

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 blindly 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 initial focus of the research presented in this chapter was to expose the issues related to weaving concerns into the C++ Standard Template Library. The study proved that adaptation has to be made not only to the template definition, but also to the application program that instantiates the template in multiple places. The key concept of weaving into template instances was experimentally validated by applying the concepts to a popular large-scale open source library for scientific computing. The scalability issues of such a requirement demanded the availability of mature parsers capable of handling several thousand lines of complex template specification.

There are several challenges that are not discussed in detail in Section 2 of the chapter, but worth mentioning here. For example, it may be possible that the base class template uses several static members (member functions or variables) in its definition and one must ensure that they remain consistent within the application program that uses instances of the class template. However, because our technique uses subtype copies of the base class template that inherits all static members from its superclass, the consistency property is still maintained. The use of assignment to enable type safety in subtypes was already discussed in Section 2.2.

Some of the challenges described in this chapter may be solved using AspectJ-like pointcuts, such as `call(vector<int>.push_back(..) && (within(A) || within(B)))`, but it does not provide additional flexibility to quantify over particular variables that refer to specific types. For example, there may be several variables of type `vector<int>` in method `M` of class `A` and it is only desired to enable logging to variable `myfoo`. Requirements like this can be elegantly handled by the approach shown in Sections 2 and 3 of the chapter, which also ensures type safety. Similarly, typedef expressions or even complex expression statements like

```
vector<int> f1s, f2s, f3s; // only required to log f1s
```

can be straightforwardly handled by this approach. In the above case, before applying any transformation, the declaration is at first broken into two parts: one that contains the variable that needs to be logged (`f1s`) and the rest that do not enable logging.

Given the tendency of concern-based template adaptation, the contribution presented in this chapter 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 [25]. Furthermore, a contribution of the chapter demonstrated the ability to modularize crosscutting concerns in scientific libraries.

The work described in this chapter is a modest initial effort that is at an early stage in terms of the construction of a concrete aspect language to cover the different situations of template instantiation (as in Table 1). The future work will involve extending the focus to other scientific libraries that are implemented in C++ (e.g., POOMA [22], MTL [24]). 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. Because of the availability of a mature FORTRAN parser within DMS, we also plan to perform aspect mining and modularization efforts

on large scale scientific applications written in FORTRAN that use scientific packages such as SCALAPACK [2]. In general, our future work will focus on modularizing High Performance Linpack (HPL) libraries designed for benchmarking high performance clusters and supercomputers.

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. L. Susan Blackford, J. Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David Walker, and R. Clint Whaley, "ScaLAPACK Users Guide," *Society for Industrial and Applied Mathematics*, 1997, (<http://www.netlib.org/scalapack/slug/>).
3. 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.
4. Luca Cardelli and Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, 17(4), December 1985, pp. 471-522.
5. Mikhail Chalabine and Christoph Kessler, "Crosscutting Concerns in Parallelization by Invasive Software Composition and Aspect Weaving," *39th Hawaii International Conference on System Sciences (HICSS-39)*, Kauai, HI, January 2006.
6. 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*, vol. 44, no. 13, October 2002, pp. 827-837.
7. Krzysztof Czarnecki and Ulrich Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.
8. 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.
9. Vladimir Getov, Susan Flynn Hummel, and Sava Mintchev, "High-performance Parallel Programming in Java: Exploiting Native Libraries," *Concurrency: Practice and Experience*, September-November 1998, pp. 863-872.
10. 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.
11. Bruno Harbulot and John Gurd, "Using AspectJ to Separate Concerns in a Parallel Scientific Java Code," *International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, March 2004, pp. 122-131.
12. 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.
13. Elizabeth Johnson and Dennis Gannon, "HPC++: Experiments with the Parallel Standard Template Library," *International Conference on Supercomputing*, Vienna, Austria, July 1997, pp. 124-131.

14. Nicolai M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, 1999.
15. 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.
16. 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.
17. Martin Lippert and Cristina Lopes, "A Study on Exception Detection and Handling Using Aspect-Oriented Programming," *International Conference of Software Engineering (ICSE)*, Limerick, Ireland, June 2000, pp. 418-427.
18. 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.
19. 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, Jan 2006, pp. 68-77.
20. JSR-000014: *Adding Generics to the Java Programming Language*, <http://www.jcp.org/aboutJava/communityprocess/review/jsr014/>
21. Daniel Quinlan, Markus Schordan, Brian Miller, and Markus Kowarschik, "Parallel Object-Oriented Framework Optimization," *Concurrency: Practice and Experience*, February-March 2004, pp. 293-302.
22. John V. W. Reynders, Paul J. Hinker, Julian C. Cummings, Susan R. Atlas, Subhankar Banerjee, William F. Humphrey, Steve R. Karmesin, Katarzyna Keahey, Marikani Srikanth, and Mary Dell Tholburn, "POOMA: A Framework for Scientific Simulations of Parallel Architectures," in Gregory V. Wilson and Paul Lu, ed., *Parallel Programming Using C++*, MIT Press, 1996.
23. Markus Schordan and Daniel Quinlan, "A Source-To-Source Architecture for User-Defined Optimizations," *Joint Modular Languages Conference (JMLC)*, Springer-Verlag LNCS 2789, Klagenfurt, Austria, August 2003, pp. 214-223.
24. 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.
25. 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.
26. 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.
27. Anthony Skjellum, Ewing Lusk, and William Gropp, "Early Applications in the Message-Passing Interface (MPI)," *The International Journal of Supercomputer Applications and High Performance Computing*, June 1995, pp. 79-95.
28. 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.

29. Todd Veldhuizen and Dennis Gannon, "Active Libraries: Rethinking the Roles of Compilers and Libraries," *SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, Yorktown Heights, NY, October 1998.
30. 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.