Weaving into Template Libraries

Suman Roychoudhury, Jing Zhang, and Jeff Gray
University of Alabama at Birmingham
1300 University Boulevard
Birmingham, AL USA 35294
[roychous, zhangj, gray] @ cis.uab.edu

ABSTRACT

Aspects have the potential to interact with many different kinds of language constructs in order to modularize crosscutting concerns. Although the initial Java-based aspect languages have demonstrated advantages of applying aspects to traditional object-oriented hierarchies, additional language concepts (e.g., parametric polymorphism) can also benefit from a synergy with aspects. Many popular languages already support parametric polymorphism (e.g., C++ templates), and other languages are soon to adopt the idea. With the acceptance of JSR-14, which brings generics to Java, investigation into the combination of aspects and generics will become more important. This paper presents a program transformation approach for weaving crosscutting concerns into template libraries. The core of the paper demonstrates the approach as applied to a large open-source C++ template library for scientific computing.

Keywords

Aspects, templates, program transformation, 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 support templates or generics (e.g., C++ and Ada). Aspects have the capability to improve the modularization of crosscutting concerns that cannot be separated otherwise in large template libraries. The ability to weave into templates offers an additional degree of adaptation and configuration beyond that provided by parameterization alone.

With the addition of generics[1] to Java in JSR-14 [18], it is expected that the application of aspects to parametric polymorphism will become a more focused research objective. However, the topic has not received much attention in the existing research literature. The most detailed discussion on the topic can be found in [17], within the context of the AspectC++ project [26]. The effort to add aspects to templates in AspectC++ has been partitioned along two complimentary dimensions: 1) weaving advice into template bodies; 2) using templates in the bodies of aspects. The initial AspectC++ work is focused on the second dimension (templates in the advice body). The contribution of our paper is a deeper investigation into the first dimension of advice weaving into the template implementation.

There are numerous issues that arise from applying AOP to parametric polymorphism. A key challenge occurs from the realization that a template is instantiated in multiple places, yet it may be the case that the crosscutting feature is required in only a subset of those instances. A language is needed to define the subset, and an appropriate copy semantics is required to separate weaved templates from the base implementation of the library. Furthermore, in addition to the transformation of the template library itself, the program that instantiates the library may also need to be altered according to the types of weaving performed on the base code of the library. These issues are discussed in detail in the paper through example case studies.

The technique presented is a source to source translation 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 [9], we applied program transformation technology to construct an aspect weaver for Object Pascal. This paper extends that work to C++ in order to address the challenges of parsing and transforming complex templates. Although the focus of the paper is on the low-level transformations, peripheral comments are made regarding the design of an aspect language that provides better abstraction of the accidental complexities of the low-level transformations pertaining to template weaving.

Scientific computing was an initial application domain for AOP [11]. However, aside from an application of AspectJ [14] to an implementation of JavaMPI [10], 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 a focus of AOP research has been applied to Java-based implementations. Nevertheless, there is a strong potential for impact if aspects can be used to improve the modularization of libraries tailored for parallel computation. Scientific computing applications written in C++ rely heavily on template libraries that are specialized for mathematical operations on vectors, arrays, and matrices [24]. Our paper makes a contribution by applying aspects to a case study of a well-known scientific computing library.

The next section contains an overview of the key concepts of template weaving, and provides a solution technique applied to a small case study. In Section 3, a popular open-source library for scientific computing serves as the context for discussion of crosscutting concerns that exist in template libraries. Section 4 provides comparison to related work. A conclusion offers summary remarks and a description of future efforts.

[1] Throughout the remainder of this paper, the term *template* is used to refer to the general concept of parametric polymorphism (even though the word *generic* is used in the Java community).

2 TEMPLATE WEAVING IN STL

This section introduces a short example that has been constructed to highlight several of the essential concepts of weaving into templates. An application of the STL vector class is presented, along with a description of a program transformation technique for weaving a crosscutting concern into vector instances.

2.1 An Introductory Example: STL Vector Class

The Standard Template Library (STL) [13] is a general-purpose C++ library that provides many data structures and algorithms (e.g., containers, iterators, algorithms, function objects and allocators). STL embraces the idea of generic programming, which describes the implementation of algorithms or data structures in a type-independent manner. This section demonstrates a technique to weave cross-cutting concerns into applications referencing STL classes, in particular, the STL vector class is chosen as an initial proof of concept. A fragment of the vector template class definition is provided in the left-hand part of Listing 1.

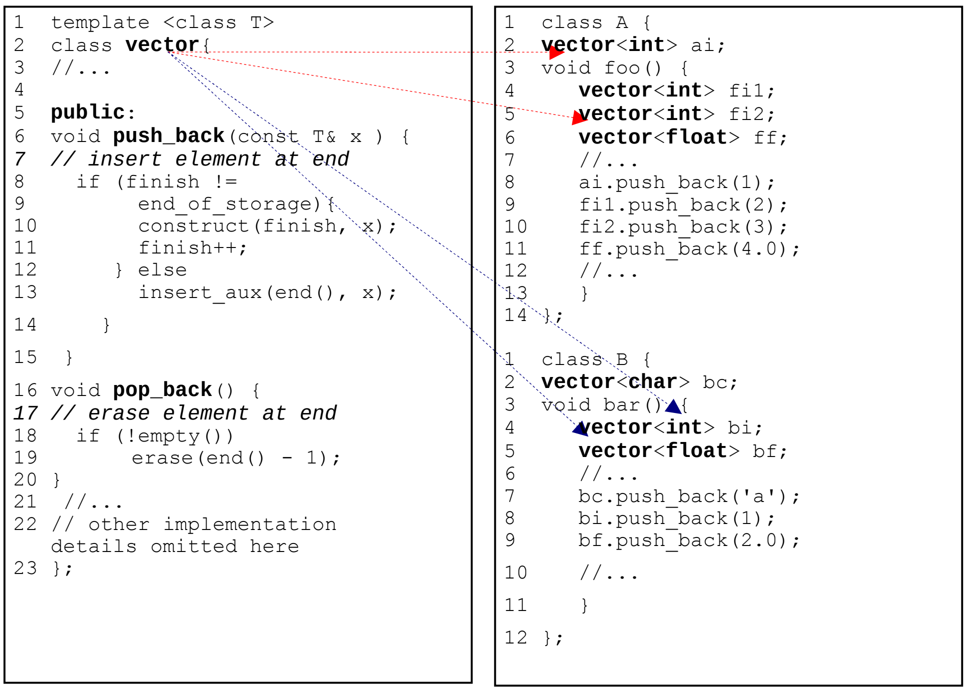
As common with any vector class definition, it provides basic operations such as, pushing an element onto the end of the vector, popping an element off the end of the vector, returning the size of the vector etc. This particular code however shows only the `push_back` method that in succession calls `insert_aux` to insert an element `x` at the end of the vector.

The sample code in the right-hand part of Listing 1 illustrates the use of a vector in an application program. In this short example, three different types of vector instances are declared (i.e., vectors of type `int`, `char`, and `float`). Each vector instance invokes the `push_back` method to insert an element. In particular, three `<int>` and one `<float>` type vector declarations are instantiated in class A while each of `<char>`, `<int>` and `<float>` vector type are declared in class B.

Considering the canonical (almost clichéd) logging example, suppose that important data in specific vector instances needs to be recorded whenever the contents are changed. Within the context of an STL vector, a requirement may state that logging is to occur for all items added to each invocation of the `push_back` method, but only for specific specializations. For example, it may be desired to log every new element of type `int` when inserted into the end of an `int` vector. In order to affect only `int` instances of vector and leave the other types (e.g., `float, double etc`) of vectors unaltered, the intuitive idea is to make a copy and rename the original vector template class (e.g., `vector_copy`).

The logging statement can then be weaved into the `push_back` method of the `vector_copy` template class. The copy of the vector class fragment is shown at the top of Listing 2.

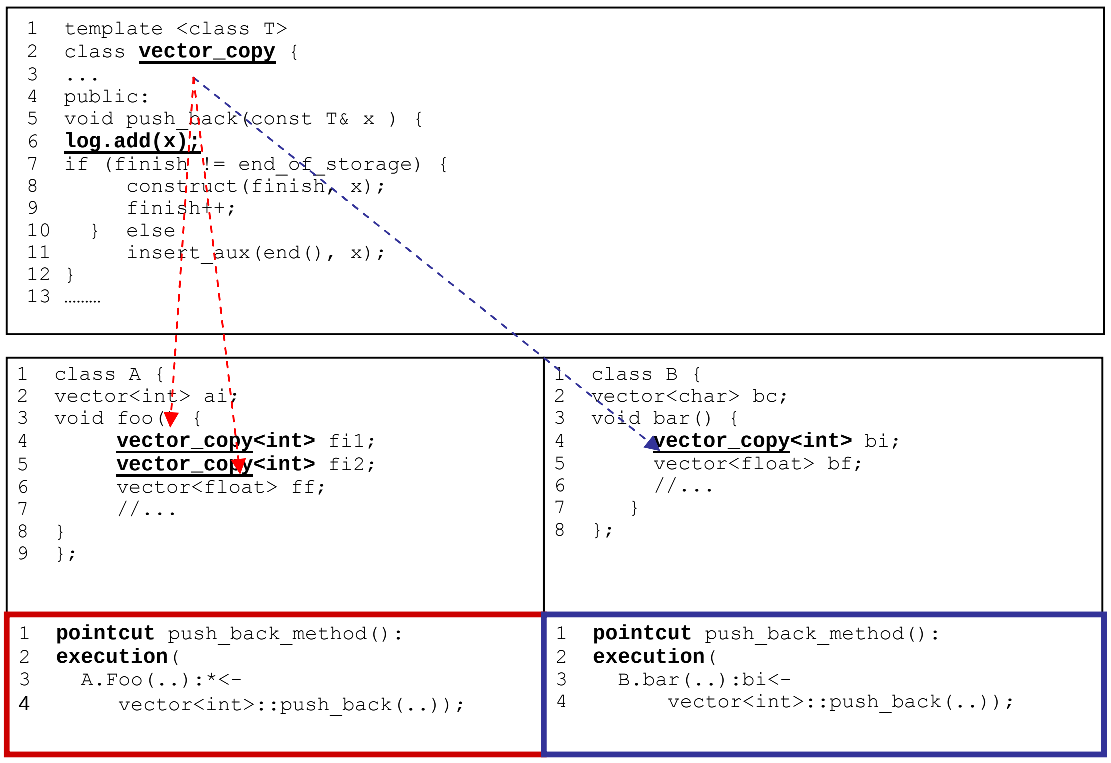
Furthermore, in addition to the library, the source code of the user application must also be updated to reference the new `vector_copy` class in the appropriate places. In this case, all of the declaration statements of `vector<int>` in the application will now reference `vector_copy<int>`. Middle of Listing 2 illustrates the corresponding changes to the user application. Note that all other vector references and specializations are left unaltered.



```
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
};
```

```
class A {
vector<int> ai;
void foo() {
   vector<int> fi1;
   vector<int> fi2;
   vector<float> ff;
   //...
   ai.push_back(1);
   fi1.push_back(2);
   fi2.push_back(3);
   ff.push_back(4.0);
   //...
   }
};

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

Listing 1. STL Vector Class and its specializations



```
template <class T>
class vector_copy {
...
public:
void push_back(const T& x ) {
log.add(x);
if (finish != end_of_storage) {
      construct(finish, x);
      finish++;
   }  else
      insert_aux(end(), x);
}
………
```

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

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

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

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

Listing 2. STL Vector copy class and updated references in the application instances

2.2 An Aspect Language for Templates

The previous section described a manual process for adding a logging concern into specific instantiations of a vector. In this sub-section, we describe the design of a higher level AspectJ-like language for weaving concerns into templates.

As discussed in section 2.1, there may be the need for multiple template instantiations to have slightly different implementations. For example, within application specific instances of the vector class there may be subtle points of variability necessary in a specific vector instance (e.g., logging as in this case). To characterize this behavior, Table 1 illustrates the scoping rules in the pointcut specification language.

Firstly, let us step back to Listing 2 to see how the newly defined scoping rules get reflected in the application instances (i.e. in class A and class B). At the bottom of that listing two pointcut specifications are shown which primarily updates the vector references in class A and class B. The pointcut on the bottom-left of listing 2 means that for method Foo defined in class A update "all" `int` vectors to capture the logging property as defined in the push_back method. The pointcut on the bottom-right of listing 2 means that for method Bar defined in class B update only the "bi" `int` vector to capture the logging property as defined in the push_back method.

Table 1. The scope representation in pointcut specifications

Designator	Description
C:*	All template instantiations within the declaration of class C, which are not local instantiations in any methods of C
* C.*(..):*	All local template instantiations within all methods of class C
(C:* || * C.*(..):*)	All template instantiations within class C
* C.M(..):*	All local template instantiations within method M of class C
* C.*(..):I	Any template instantiation that is named I, in all methods of class C
* C.M(..):I	Local template instantiation I, in method M of class C

To show further, how this new scoping rule can affect the weaving of the logging concern as described in section 2.1, a few more examples are provided in Listings 3 through 7. 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 3 demonstrates an example of the aspect language to add the logging statement to the push_back method in all vectors of any type. The pointcut push_back_method() represents the points of execution where weaving is to occur. Vector<*> is used to denote that weaving is performed on all types of vector instances. Therefore, for this aspect specification, the corresponding low level implementation (i.e., by using DMS rule specification language) would make a copy of the whole vector class definition (e.g., named as vector_copy), and insert the log.add(x) statement at the beginning of its push_back method. Correspondingly, every reference to the original vector instantiation in the application program will now reference the vector_copy instantiation.

```
aspect InsertPushBackLogToAllVector {

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

  before():push_back_method() {
    log.add(x);
  }
}
```

Listing 3. Aspect specification for inserting the push_back log to all vectors of ANY type in ANY class

Listing 4 defines a pointcut that specifies the execution join point for the push_back method of all vectors of type int. The low level implementation involving DMS Rules can capture this same intention and will be shown in the following section.

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

Listing 4. Pointcut specification for weaving into all vectors of type int in ANY class

To add finer granularity, Listing 5 describes the pointcut specification for weaving into all vectors of type int in class A. To be more specific in limiting the scope of a pointcut, Listing 6 defines a pointcut capturing all int vectors in method Foo that are in class A. Listing 7 is the most specific pointcut; it will only weave into a particular instance variable fi1 whose type is an int vector.

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

Listing 5. 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 6. Pointcut specification for weaving into all vectors of type int in class A::Foo

```
pointcut push_back_method():
  execution(
    * A.Foo(..):fi1<-
              vector<int>::push_back(..));
```

Listing 7. Pointcut specification for weaving into int vector fi1 in class A::Foo

2.3 Weaving Concerns into STL – An Automated approach using DMS

The aspect specification shown in the previous section forms the high level specification language to perform the weaving. In this sub-section we will demonstrate the low level implementation details used to automate the weaving process by using a program transformation engine, namely the Design Maintenance System.

2.3.1 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 teminology, 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 (concrete syntax) defined by a language domain. Typically, a large collection of RSL files, like those represented in Listing 8 and Listing 9, are needed to describe the full set of transformations. Patterns describe the form of a syntax tree. They are used for matching purposes to find a syntax tree having a specified structure. Patterns are often used on the right-hand side (target) of a rule to describe the resulting syntax tree after the rule is applied. 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.

In addition to the RSL, a language called PARLANSE 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 that provided by RSL transformations rules. PARLANSE is a functional language whose programs can be tied to 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 while the aspect specification represents a higher level of abstraction that hides the unnecessary details (i.e., the primitive transformations are too low-level for widespread adoption) from

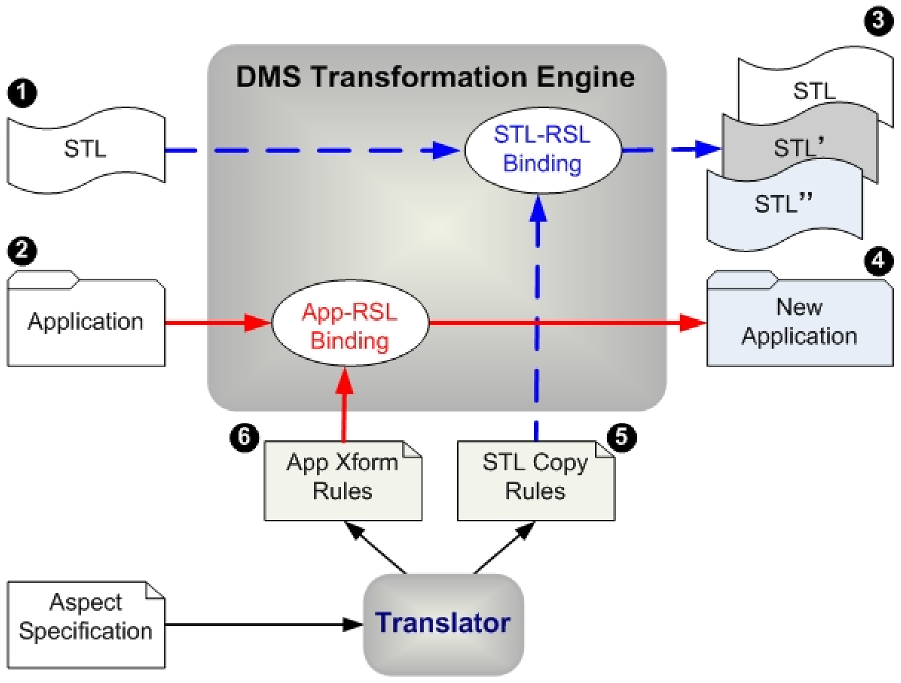


Figure 1. Overview of Template Weaving Process

the underlying implementation. Figure 1 presents an overview of an automated process for template weaving. As shown in the bottom of this figure, the low-level rules along will their corresponding binding with the higher-level aspect specification will act on the target STL source to necessitate the behavioral changes in them.

Two core engines are involved in the implementation: one is the translator, whose intent is to parse and translate a high-level aspect language into low-level transformation rules (i.e., item numbers 5 and 6); another is the DMS transformation engine, which will take the source files and the generated rules as input, and transform the source files based on the rule specifications. 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 (shown as item #2), and an AspectJ-like language specification (examples shown in Section 2.2) that is used to describe the specific pointcuts and advice for template weaving. The translator engine includes a lexer, parser, 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 pattern instantiation process is similar to our previous work on building an aspect domain for ObjectPascal [9]. The STL copy rules make a copy of the original STL template and weaves in the new concerns by use of the STL-RSL Binding in the transformation engine. As a result, several different copies of STL templates will be generated, each of which has one specific concern weaved into its base definition (shown as #3). Likewise, 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 along with the generated STL copies.

The remaining parts of Section 2 will introduce a detailed discussion of transformation rules used for implementing the template weaving concepts.

2.3.2 *Transformation Rules for Template Weaving*

Listing 8 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 copy of the vector template class, and insert the logging statement into appropriate placeholders in the abstract syntax tree. The first line in the rule snippet establishes the default base language domain to which the DMS rules are applied (in this case, Visual C++ 6.0 is used). Pattern `log_statement` in lines 3 and 4 represents the log statement that will be inserted before the execution of the `push_back` method. Pattern `weaved_method_name` in lines 6 and 7 defines the name of the method that will be transformed (i.e., `push_back` in this case). Pattern `new_template_name` in lines 9 and 10 specifies the new name for the copied vector (e.g., `vector_copy`). In DMS, exit functions (i.e. external patterns and functions) are written in PARLANSE, which is a parallel language for symbolic expressions. It provides an enriched set of API's for performing various operations on the abstract syntax tree. In this example the external pattern `copy_template_add_log_to_pushback_method` is implemented in PARLANSE cwhich does the actual process of copying, renaming, and weaving. The 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 copy. Due to limited space, the PARLANSE code is omitted here (all of the transformation code, papers, and videos are available on the GenAWeave project web page at http://www.cis.uab.edu/gray/Research/GenAWeave). The rule `insert_log_to_template` on line 22 triggers the transformation on the vector class by invoking the specified external pattern. Notice that there is a condition associated with this rule (line 30), which describes a constraint stating that the rule should be applied only once. After applying this rule to the code fragment `(i.e.,vector.h)` shown in Listing 1, a new template class named `vector_copy` 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 2).

However, due to the low level nature of the rule specification language, software developers are not expected to write rules in this manner, rather the aspect language mentioned in section 2.2 and its corresponding binding with the RSL would drive the weaving. Developers can specify the join point and pointcut using the aspect specification and underlying implementations would be then instantiated oblivious to them.

```
1  default base domain Cpp~VisualCpp6.
2
3  pattern log_statement():
4    statement_seq = "log.add(x);".
5
6  pattern weaved_method_name():
7    identifier = "push_back".
8
9  pattern new_template_name():
10   identifier = "vector_copy".
11
12 external pattern
13   copy_template_add_log_to_pushback_method
14     ( td : template_declaration,
15       st : statement_seq,
16       method_name : identifier,
17       template_name : identifier ):
18   template_declaration =
19   'copy_template_add_log_to_pushback_method'
20   in domain Cpp~VisualCpp6.
21
22 rule insert_log_to_template
23   ( td : template_declaration ):
24   template_declaration ->
25   template_declaration
26 = td               ->
27   copy_template_add_log_to_pushback_method
28   (td, log_statement(),
29   weaved_method_name(),new_template_name())
30 if td ~=
31    copy_template_add_log_to_pushback_method
32    (td, log_statement(),
33   weaved_method_name(),new_template_name()).
34
35 public ruleset applyrules =
36   { insert_log_to_template  }.
```

Listing 8. DMS transformation rules for weaving log statement into `push_back` method

The weaving process is still not complete as the application program also needs to be updated to reference to this new vecor_copy instance. The DMS transformation rule specification for updating the corresponding application program is specified in Listing 9. Pattern `pointcut` (line 3) identifies the condition under which the rule will be applied (e.g., all `int` vector declarations). Pattern `advice` (line 7) defines the name of the new transformed function (e.g., `vector_copy`). The external pattern `replace_vector_instance` performs the actual transformation implementation in PARLANSE. After applying this particular rule to a user application, every instance vector declared as `int` will be transformed into an instance of `vector_copy`.

```
1  default base domain Cpp~VisualCpp6.
2
3  pattern pointcut( id : identifier ):
4    declaration_statement =
5      "vector<int> \id;".
6
7  pattern advice( id : identifier ):
8    declaration_statement =
9      "vector_copy<int> \id;".
10
11 external pattern replace_vector_instance
12   ( cd  : class_declaration,
13     ds1 : declaration_statement,
14     ds2 : declaration_statement ):
15   class_declaration =
16     'replace_vector_instance'
17   in domain Cpp~VisualCpp6.
18
19 rule replace_template_instance
20   ( cd : class_declaration,
21     id : identifier):
22   class_declaration ->
23   class_declaration
24 = cd  ->  replace_vector_instance
25            (cd,pointcut(id),advice(id))
26 if cd ~=  replace_vector_instance
27            (cd,pointcut(id),advice(id)).
28
29 public ruleset applyrules =
30   { replace_template_instance }.
```

Listing 9. DMS transformation rules for updating the application program

2.4 An application using STL – Internet Communications Engine

In this subsection we concentrate on verifying the practical usage of the basic concepts developed in the previous section by applying it to an open source application, namely the Internet Communications Engine (ICE). ICE is an object-oriented middleware platform suitable for use in heterogeneous environments and supports developments of practical distributed applications for a wide variety of domains. It provides tools, APIs and libraries for building object-oriented client server applications. The reason ICE was chosen to validate the concepts developed in section 2.1 was primarily due to its extensive usage of the C++ standard template library. The ICE core package consists of several hundred C++ classes and is around 155 K SLOC in size.

Although there are several places in the core ICE library that are suitable candidates for applying aspect-oriented template specializations, here we only illustrate the BasicStream class. The BasicStream class consist of several readers and writers and is instrumental in implementing several data types (int, float, char, long, double, string etc) for communicating between the object streams. Obviously the objects need to be type casted to bytes before being transmitted as streams. Although this kind of type casting operation is not considered as a traditional aspect (contrary to logging, error-handling etc) in the AOP world, but the fact that this operation cross-cuts the entire stream class cannot be ignored. The following piece of code exemplifies the problem.

```
1  ...
2  IceInternal::BasicStream::write(const
3                vector<Float>& v) {
4  Int sz = static_cast<Int>(v.size());
5  #ifdef ICE_BIG_ENDIAN
6    const Byte* src = reinterpret_cast
7    <const Byte*>(&v[0]) + sizeof(Float) - 1;
8  // writing to the byte stream (omitted)
9  #else
10    memcpy(&b[pos], reinterpret_cast<const
11         Byte*>(&v[0]), sz * sizeof(Float));
12 #endif
13 }
14 ...
```

Listing 10. Type casting operation for non byte data types

Line 4 and Line 6 illustrates the two cross-cutting concerns which are scattered throughout the code base and appears 9 times each for every read and write operation. ICE uses its own user defined Int data type and needs to know the size of the vector before converting it to byte streams (Line 4 shows this case). Both these operations could be abstracted as an aspect and could be present as a new method in the parent template class. To weave these kinds of cross-cutting properties, we would use the inter type declaration mechanism (similar to inter type declaration in AspectJ), and the code in Listing 11 show the aspect specification to do this.

```
1  aspect size_typecast {
2    public Int vectorSize() {
3       return static_cast<Int>(v.size());
4    }
5    public const Byte* srcFloat() {
6       reinterpret_cast<const
7       Byte*>(&v[0]) + sizeof(Float) - 1;
8      }
9    public const Byte* srcDouble() {
10      reinterpret_cast<const
11      Byte*>(&v[0]) + sizeof(Double) - 1;
12   }
13      . . .
14 }
```

Listing 11. Aspect Specification for weaving size and type cast operation

Note that the reference in Line 3 and 5 in the source application should also be updated to point to this new method declaration within the parent template.

3 ASPECTS IN SCIENTIFIC LIBRARIES

The previous section presented examples that were contrived to illustrate some of the distinct scoping problems that are associated during weaving into specific template specializations. In this section, focus is given towards modularizing open-source libraries written in the scientific computing domain. The contribution highlights the crosscutting features on a real case study, and describes improved modularization using the template weaving idea. In particular, this section focuses on identified aspects in Blitz++ [28], a C++ template library that supports high performance scientific computing.

3.1 Background: Crosscutting in Blitz++

Optimizing performance, while preserving the benefits of programming language abstractions [27, 24, 19], is a major hurdle faced in scientific computing. 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++ [12] and High Performance Java [8]) and scientific libraries (e.g., POOMA [20], MTL [22], and BLITZ++ [28]) 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 and generics 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 section 2. 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 majority of the section demonstrates the weaving technique using DMS that was introduced in Section 2. The general approach could be applied to other languages that support parametric polymorphism (e.g., Ada and Java 1.5).

The first example (Section 3.2) 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`. An additional 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 second example (Section 3.3) 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 example is similar in concept to the redundant assertions that were described by Lippert and Lopes in [16].

Section 3.4 examines AOP combined with other generative programming techniques [6]. In particular, the section explores the various binary and unary operations on vectors that use templatized 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 3.4, over 12K SLOCs are generated using just 14 lines of code in a base template. We have identified six additional aspects in the Blitz++ library, but space limitations prohibit a complete discussion.

3.2 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 runs very slowly primarily because Blitz++ does several 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 12 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 cuts across 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 12) 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
22      setupStorage(2);
23 }
...
```

Listing 12. Precondition check and setpupStorage in array implementation

With respect to the aspect language design presented in Section 2.2, Listing 13 contains a simple aspect specification for the cross-cutting concern shown in Listing 12. The expression statements in `BZPRECONDITION` and `setupStorage` form part of the before and after advice. The pointcut refers to all `Array` constructors within the `Array` implementation class. The function call `tjp.getArgs().length()` will return the length of the parameter list in the `Array` constructor. This is a special construct which is implemented internally by the PARLANSE external function `generate_paramlist_length` defined in Listing 15.

```
aspect InsertBZPreCondition_MemAllocation {

  pointcut ArrayConstuctor():
    execution(Array<*>::Array(..));

  before(): ArrayConstuctor()
    {  BZPRECONDITION(N_rank
          >= tjp.getArgs().length());}

  after(): ArrayConstuctor()
    {  setupStorage(
              tjp.getArgs().length()-1);}}
}
```

Listing 13. Aspect specification for precondition and memory allocation in templates

The low level RSL implementation used to weave these features is contained in Listing 14. The parameterized rules `insert_BZPRECONDITION` and `insert_SetupStorage` are used to insert the before and after advice for the array implementation. Note that there is a marked similarity between the two rules, which could be abstracted to form a single rule. However, the `BZPRECONDITION` statement is attached before the body of the array implementation, whereas the `setupStorage` statement is attached after the main body. This is achieved by the patterns `BZPRECONDITIONAspect` and `SetupStorageAspect`. The exit functions (i.e., external patterns and conditions) are coded in PARLANSE and are used to compute the value for the storage and rank parameters. This value is derived from the parameter list of the array constructor (e.g., Line 5 in Listing 12) as shown in the external function `generate_paramlist_length` in Listing 15. The `expression_list` is generated according to the number of parameters. If the first parameter type is `int`, and array length is 2, expression statements like (`N_rank >= 2`) and `setupStorage(1)` will be weaved. Part of the code is commented in italics for easy readability.

The strategy to count the numbers of arguments in a parameter list is shown in the listing below. The primary search criterion is based on walking the abstract syntax tree seeking for parameter declaration list nodes. On successful match, all the children belonging to the list are enumerated one by one and the counter is incremented. Finally the total length of the argument list returned to the caller.

```
default base domain Cpp~VisualCpp6.

pattern BZPRECONDITION ():
  identifier_or_template_id = BZPRECONDITION".

pattern BZPRECONDITIONStmt
  (p : para_decl_clause):
  expression_statement =
  "\BZPRECONDITION \(\)( N_rank >=
   \ generate_paramlist_length \(\p\));".

-+ Before advice
pattern BZPRECONDITIONAspect
  (p : para_decl_clause, s : statement_seq):
  compound_statement =
  "{\BZPRECONDITIONStmt\(\p\) { \s }} ".

pattern SetupStorage ():
  identifier_or_template_id = "setupStorage".

pattern SetupStorageStmt(p: para_decl_clause):
  expression_statement =
  "\SetupStorage \(\)
  (\generate_paramlist_length \(\p\));".

-+ After advice
pattern SetupStorageAspect
  (p : para_decl_clause,
   s : statement_seq):
  compound_statement =
  "{ \s \SetupStorageStmt\(\p\) } ".

external pattern
  generate_paramlist_length
  (p : para_decl_clause):
  INT_LITERAL =
  ' generate_paramlist_length '
  in domain Cpp~VisualCpp6.

external condition
  para_type_is_int(p : para_decl_clause) =
  'para_type_is_int'
  in domain Cpp~VisualCpp6.

rule insert_SetupStorage
  (p : para_decl_clause,
   c : ctor_initializer,
   s : statement_seq):
  function_definition -> function_definition
= "Array (\p) \c { \s } " ->
  "Array (\p) \c
     {\SetupStorageAspect\(\p \, \s\) }"
if ~[modsList:statement_seq .s matches
   "\:statement_seq \SetupStorageAspect\(\p \,
   \modsList\)"] and para_type_is_int(p).

rule insert_BZPRECONDITION
  (p : para_decl_clause,
   c : ctor_initializer,
   s : statement_seq):
  function_definition -> function_definition
= "Array (\p) \c { \s } " ->
  "Array (\p) \c
     {\ BZPRECONDITIONAspect \(\p \, \s\) }"
if ~[modsList:statement_seq .s matches
   "\:statement_seq \ BZPRECONDITIONAspect
   \(\p \, \modsList\)"] and
   para_type_is_int(p).

public ruleset applyrules =
{
   insert_BZPRECONDITION,
   insert_SetupStorage
}.
```

Listing 14. Weaving precondition and storage allocation using RSL

```
(define generate_paramlist_length
(lambda Registry:CreatingPattern
   (value (local (;;
                  // local variable declaration omitted
                  );;
(;;
  (= para_decl_list
     (AST:GetFirstChild arguments:1))
 // fetch the last parameter from the array parameter list
 (while (== (AST:GetNodeType para_decl_list)
             _pp_decl_list)
    (;; (+= n) //increase parameter count
        (= para_decl_list
           (AST:GetFirstChild para_decl_list)
    );;
  )while
  (= literal_num
  (AST:CreateNode rep_instance _int_literal))
   // set the parameter count n to the literal  value node
  (AST:SetNatural literal_num n)
  (return literal_num)
);;
...
)lambda
)define
```

Listing 15. PARLANSE function to generate the conditional parameter value

3.3 Redundant Assertion Checking

Another debugging feature present in Blitz++ checks for the size or range of the arrays. For example, Listing 16 shows a 4x4 array instantiation and subsequent allocation of floating point values to an array index. However, because this is a C-style array, the valid index ranges are 0..3 and 0..3; hence, it is an error to refer to an invalid index.

```
int main()
{
   Array<complex<float>, 2> Z(4,4);
   Z(4,4) = complex<float>(1.0, 0.0);
   return 0;
}
```

Listing 16. Accessing an invalid array index

```
_bz_bool assertInRange(int BZ_DEBUG_PARAM(i0),
         int BZ_DEBUG_PARAM(i1)) const
{
  BZPRECHECK(isInRange(i0,i1),
     "Array index out of range: ("
     << i0 << ", " << i1 << ")"
     << endl << "Lower bounds: "
     << storage_.base() << endl
     << "Length: " << length_ << endl);\
  return _bz_true;
}
```

Listing 17. Definition of the assertion function

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 17 shows the internal implementation of the `assertInRange` function in Blitz++.

This particular assertion is defined in all array template specifications, according to the general pattern shown in Listing 18 (the `assertInRange` aspect in Lines 5 and 12). 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 18, 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 specification is repeated approximately 46 times in array-impl.h and is context-dependent on the size of each template container.

```
template<int N_rank2>
T_numtype operator()
     (TinyVector<int,1> index) const
{
  assertInRange(index[0]);
  return data_[index[0] * stride_[0]];
}

T_numtype operator()
     (TinyVector<int,2> index) const
{
  assertInRange(index[0], index[1]);
  return data_[index[0] * stride_[0] +
               index[1] * stride_[1]];
}
...
```

Listing 18. Redundant assertion check on base template specification

To avoid the crosscutting assertion checking in every definition of array implementation, the aspect specification (as defined in Listing 19) will weave this concern back into the template code. The operation_func pointcut in this specification refers to all operation constructors in the array implementation class. The special construct tjp.getParameterList is internally mapped to a local parlanse function which returns part of the valid AST structure observed at this join point.

```
aspect AssertInRange {
  pointcut operator_func():
    execution(Array<*>::operator()(..));

  before(): operator_func()
    {
     assertInRange(tjp.getParameterList());
    }
```

Listing 19. Aspect specification for redundant assertion checks

In the current effort, aspect mining and removal of the original crosscutting features was performed manually, although the actual weaving back into the base code is automated with the illustrated transformations. Future work will explore aspect mining and clone detection within the context of templates, but is not a focus of this paper.

The low level RSL implementation of the above aspect is shown in Listing 20. The pattern assertInRangeAspect takes three parameters; namely, the integer literal that specifies the size of the TinyVector, the vector identifier (i.e., index), and the statement sequence node where the aspect is to be inserted. The expression statement pattern assertInRangeStmt is generated by an externally defined pattern (i.e., create_expression_list_for_vectorIds), which uses an internal PARLANSE specification to generate the parameters for the assertInRange function.

```
default base domain Cpp~VisualCpp6.

pattern vectorIdPattern(vectorId: identifier):
  identifier = "\vectorId".

external pattern
  create_expression_list_for_vectorIds
  (n: INT_LITERAL, vectorIdRoot: identifier):
  expression_list =
  'create_expression_list_for_vectorIds'
  in domain  Cpp~VisualCpp6.

pattern
 assertInRange_as_id_or_template_id():
 identifier_or_template_id = "assertInRange".

pattern assertInRangeStmt
  (n:INT_LITERAL, vectorId: identifier):
  expression_statement =
  "\assertInRange_as_id_or_template_id\(\)
  (\create_expression_list_for_vectorIds
   \(\n \,\vectorIdPattern\(\vectorId\)\));".

pattern assertInRangeAspect(n : INT_LITERAL,
  vectorId: identifier, s: statement_seq):
  compound_statement =
  "{\assertInRangeStmt\(\n\,\vectorId\){\s}}".

rule insert_assertInRange
  (n : INT_LITERAL, vectorId : identifier,
   c : cv_qualifier_seq, s : statement_seq):
  function_definition -> function_definition
= "T_numtype operator()
   (TinyVector<int,\n> \vectorId) \c { \s } "
->"T_numtype operator()
   (TinyVector<int,\n> \vectorId) \c
     {\assertInRangeAspect\(\n \,
      \vectorId \, \s\)}"
if ~[modsList:statement_seq .s matches
   "\:statement_seq \assertInRangeAspect\
   (\n \, \vectorId \, \modsList\)"].

public ruleset applyrules =
  {insert_assertInRange}.
```

Listing 20. RSL specification showing weaving of assertion check to base implementation

3.4 Crosscutting Generic Functions

This sub-section discusses the combination of AOP with other generative programming techniques [6]. 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 sub-section, attention is focused on a specific set of unary vector operations in a template specification, which are generated to the vecuops.cc source file in the Blitz++ library. The vecuops.cc file is around 12K SLOCs in size; however, most of the mathematical operations (e.g., log, sqrt, sin, floor, fmod) have the same syntactic pattern structure that can be specified concisely. An analysis of the generation process revealed that the entire template specification is essentially a cross-product between the set of defined mathematical functions (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.

The mathematical functions supported in Blitz++ vectors can be enumerated as $\lambda_1, \lambda_2, \ldots \lambda_n$, and the base routine representing the pattern template structure as β (Listing 21). The code generated as the cross-product would be $\lambda_1\beta + \lambda_2\beta + \lambda_3\beta$

(i.e., { $\lambda_1, \lambda_2 .. \lambda_n$ } $\times$ β). The partial string identifier OPERATION (highlighted in bold in Listing 21), represents the locations in the pattern structure where the mathematical methods need to be weaved to generate the entire template specification (i.e., the 12k SLOCs in the `vecuops.cc` file). The concept here is somewhat different than standard AOP practice, but the idea of a cross-product between a set of mathematical functions and a base pattern is germane to the overall process of template weaving. Although the example provided in this sub-section is based on vector operations using mathematical functions, similar situations exist in several other generated template specifications in the Blitz++ library.

```
template<class P_numtype1>
inline
_bz_VecExpr<_bz_VecExprUnaryOp<VectorIterConst
  <P_numtype1>,_bz_OPERATION<P_numtype1> > >

OPERATION(const Vector<P_numtype1>& d1)
{
  typedef bz_VecExprUnaryOp<VectorIterConst
    <P_numtype1>,_bz_OPERATION<P_numtype1>>
  T_expr;

  return
    _bz_VecExpr<T_expr>(T_expr(d1.begin()));
}
```

Listing 21. Subset of base pattern used to generate the vector operation template

```
1  default base domain Cpp~VisualCpp6.
2
3  pattern aspect_op():identifier = "OPERATION".
4  pattern aspect_bz_op():
5          identifier ="_bz_OPERATION" .
6
7  pattern operation1(): identifier ="abs".
8  pattern operation2(): identifier ="acos".
9  pattern operation3(): identifier ="acosh".
10 ...
11
12 external pattern
13   search_aspect_generate_template_code
14   (td : template_declaration,
15    id1: identifier,
16    id2: identifier):
17   template_declaration =
18   'search_aspect_generate_template_code'
19   in domain Cpp~VisualCpp6.
20
21 rule generate_vec_template
22   (td:template_declaration):
23   declaration_seq -> declaration_seq
24 = td ->
25   search_aspect_generate_template_code(td,
26   aspect_op(), aspect_bz_op(), operation1(),
27   operation2(), operation3())
28 if td ~=
29   search_aspect_generate_template_code(td,
30   aspect_op(), aspect_bz_op(), operation1(),
31   operation2(), operation3()).
32
33 public ruleset applyrules =
31 {
32   generate_vec_template
33 }.
```

Listing 22. Rules showing the concept of applying AOP with generative programming

The RSL rule describing the weaving of the mathematical functions with the standard routine is shown in Listing 22. The right-hand side of the rule specification is an external pattern (i.e., `search_aspect_generate_template_code` in Line 25) that takes a list of input parameters. The first parameter is the template pattern definition (β). The second and the third parameter are the two markers in the base tree that need to be replaced with the enumerated mathematical operations. The fourth and subsequent parameters are the set of generic mathematical functions (`operationX()`) to be weaved into the base pattern.

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 [17]. 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 chose DMS for our experimentation because of our assurance of the ability to parse 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++. The current publically available version of AspectC++ (0.9pre1) is not able to parse templates, let alone the complexity found in Blitz++ or the STL.

As an alternative to DMS, there are several other transformation systems that are available (e.g., ASF+SDF [3], TXL [5]) and 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 [7]. In similar work, Lämmel [15] discusses the implementation of an aspect weaver for a declarative language using functional meta-programs.

Within the scientific computing domain, ROSE provides optimizations using source to source transformation of ASTs for C++ applications [19]. The transformations are expressed using a domain-specific language [21]. The type of transformations performed by ROSE are focused solely on optimization of scientific libraries, and are not applicable to the kinds of transformations we advocate in this paper to improve the modularization of crosscutting concerns.

5. CONCLUSION

Parametric polymorphism provides 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 are in need of 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 instances of a template. A capability is needed to identify and specify those instances that are affected by an aspect, and to provide appropriate source transformations that make a copy of the original template and weave on each copy.

The initial focus of the research presented in this paper was to expose the issues related to weaving concerns into the C++

standard template library. The study proved that the 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. However, there remain several limitations and open questions to be answered. For example, what happens if there are static variables in the template definition? Obviously a blind copy would not work there, so certain static analysis needs to be done to adapt the approach for such cases. There needs to be further study and research in this area and this paper is the first of its kind to raise concerns and issues related to weaving aspects into template instances.

Given the tendency 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 Java 1.5, as discussed by Silaghi [23]. Furthermore, a contribution of the paper demonstrated the ability to modularize crosscutting concerns in scientific libraries.

The work described in this paper is a modest initial effort that is at an early stage in terms of the construction of a concrete aspect language to cover all of the different situations of template instantiation (as in Table 1). We have developed all of the low-level transformation rules and associated PARLANSE binaries to implement all of the crosscutting examples presented in Sections 2 and 3. The future work will involve extending the focus to other scientific libraries that are implemented in C++ (e.g., POOMA [20], MTL [22]). An interesting topic that we will investigate is *library-independent* aspects that may exist within a specific domain, such as scientific computing. Because of the availability of a mature FORTRAN parser within DMS, we plan to perform aspect mining and modularization efforts on large scale scientific applications written in FORTRAN that use scientific packages such as SCALAPACK [2]. Our collaborators on this future work will be researchers at the High Performance Computing laboratory at UAB, who helped to pioneer standards within scientific and parallel programming (such as the MPI standard [25]).

ACKNOWLEDGEMENTS

Thanks to Carl Wu for fruitful discussions on the "<-" operator of Section 2.3.

4 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. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. 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] James Cordy, Thomas Dean, Andrew Malton, and Kevin Schneider, "Source Transformation in Software Engineering using the TXL Transformation System," *Special Issue on Source Code Analysis and Manipulation, Journal of Information and Software Technology (44, 13)*, October 2002, pp. 827-837.

[6] Krzysztof Czarnecki and Ulrich Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[7] 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.

[8] 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.

[9] Jeff Gray and Suman Roychoudhury, "A Technique for Constructing Aspect Weavers Using a Program Transformation System," *International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, March 22-27, 2004, pp. 36-45.

[10] Bruno Harbulot and John Gurd, "Using AspectJ to Seperate Concerns in a Parallel Scientific Java Code," *International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, March 22-27, 2004, pp. 122-131.

[11] 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.

[12] Elizabeth Johnson and Dennis Gannon, "HPC++: Experiments with the Parallel Standard Template Library," *International Conference on Supercomputing*, Vienna, Austria, July 1997, pp. 124-131.

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

[14] 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.

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

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

[17] 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.

[18] *JSR-000014: Adding Generics to the Java Programming Language*, http://www.jcp.org/aboutJava/communityprocess/review/jsr014/

[19] Daniel Quinlan, Markus Schordan, Brian Miller, and Markus Kowarschik, "Parallel Object-Oriented Framework Optimization," *Concurrency: Practice and Experience*, February-March 2004, pp. 293-302.

[20] 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 Gregory V. Wilson and Paul Lu, ed., *Parallel Programming Using C++*. MIT Press, 1996.

[21] 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.

[22] Jeremy G. 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] Todd L. 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.