MAKING THE USAGE OF STL SAFE

Douglas Gregor and Sibylle Schupp
Dept. of Computer Science
Rensselaer Polytechnic Institute
Troy, NY USA
{gregod, schupp}@cs.rpi.edu

Abstract The use of the C++ Standard Template Library has many advantages, but comes with a unique set of problems that have not been addressed by compilers or development tools. Many of these problems are due to misuses of the STL components and algorithms that are syntactically correct but semantically flawed. We motivate the case for the use of static analysis to diagnose such problems and describe our approach to checking STL usage with an STL-aware static analysis.

Keywords: Static program analysis, algorithm specification, Standard Template Library (STL), symbolic loop analysis

1. Introduction

The introduction of the Standard Template Library [14, 10] into the C++ language [3] brought generic programming into mainstream C++ programming. Increasingly, the C++ community has embraced the so-called "STL style" programming methodology, which favors high-level components over low-level language features and uses iterators to decouple the algorithmic view of data from their underlying storage representation. While this new "STL style" has advantages in terms of reusability, readability, and maintenance, it comes with a unique set of problems that are not adequately addressed by current compiler or development tool technology. Perhaps the most obvious of these problems is the deluge of error messages an error as trivial as a syntactically incorrect template instantiantion can trigger. It is not uncommon for a single-character typing error to result in a thousand or more lines of error messages, often referring to functions and types that the user has never directly used. These syntactic errors during template instantiation have been partially addressed by concept checking [13] and static "is-a" checks [8],

```
int transmogrify(int x);
vector<int> values; // fill values
vector<int> results;
transform(values.begin(), values.end(),
          results.end(), transmogrify);
```

Figure 1. A common error in the use of STL is to assume that one can insert values into a container by specifying the end iterator as the receiver of an algorithm's output.

and also by the compiler- and library-specific replacement of expanded type names with shortened versions [17].

However, syntactic errors do not dominate development costs. Semantic errors that pass unnoticed through compilers cost a great deal more in debugging and maintenance time and should be checked as well, especially since many user problems with STL result from a direct misuse of STL components and algorithms. In illustration, Figure 1 (due to Meyers [9]) shows a common error in the use of STL algorithms that output their results via output iterators. In this code fragment the user's intention is clear: `transmogrify` each of the elements in `values` and insert them at the end of `results`. Since it is illegal, however, to dereference or increment the end iterator of a container and therefore illegal to write using it, such code is always semantically incorrect. To our knowledge, this kind of error will trigger neither warning nor error from any existing compiler, and at run time the behavior will be unpredictable at best.

While the above example may be dismissed as the mistake of a novice, deeper problems lurk in the use of the STL. Perhaps the most perplexing of these problems are due to iterator invalidation, wherein a previously dereferenceable or past-the-end iterator becomes singular due to an operation on the container it references. Figure 2 illustrates one such error, where the user is attempting to perform an insertion into a sorted vector and to return an iterator to the newly-inserted element. Here, the problem is that the vector insertion might cause the vector to reallocate its storage, which, depending on whether or not the reallocation takes place, will or will not cause the iterator returned to be singular. Unfortunately, this bug will manifest itself under very specific conditions, which furthermore vary with the STL platform. Such misuse of iterators is at the same time left undetected by all current compilers and we know of only one attempt to address the problem: the "debug mode" of the STLport implementation that can detect at run time whether the user has attempted to use an invalid iterator. While this check is a marked

```
vector<int>::iterator
insert_in_sorted_vector(vector<int>& values, int val)
{
  vector<int>::iterator lb =
    lower_bound(values.begin(), values.end(), val);
  values.insert(lb, val);
  return lb;
}
vector<int>::iterator pos = insert_in_sorted_vector(v, 42);
v.insert(pos, 41);
```

Figure 2. Elusive errors in STL usage often stem from iterator invalidation; here the iterator returned from the function may be singular, because the vector insertion may cause reallocation of the vector.

improvement over the existing support, it suffers from the limitations inherent to all run-time checks. For one, it will not detect errors such as the one of Figure 2 unless specific conditions for the input parameters hold. Second, it incurs a performance penalty due to the extra code generated and executed. In contrast, static analysis follows all paths of execution at compile time, and thus catches errors that may only occur under specific conditions. Static analysis also exacts no performance penalty at run time because it does not require the addition of code into the program to check correctness. For these reasons, every modern compiler uses static analysis in its semantic validation pass, to report user errors such as the use of variables before they are initialized or to warn about unreachable code.

In this paper we apply and extend the techniques of static analysis and check the misuse of iterators by means of an STL-aware static analysis of C++ code. We first justify the need of an STL-aware static analysis in lieu of the traditional static analysis in Section 2 and illustrate the results our static analyzer produces in Section 3. We then discuss the implementation of the analyzer in Section 4 and conclude with a discussion of future challenges in checking the use of STL in Section 5.

2. Why Should the Analysis be STL-Aware?

Why should the compiler's static analysis be aware of the existence of the Standard Template Library if the STL itself is merely a collection of ordinary C++ code that enjoys no more rights or privileges than the user's C++ code? It is known that existing static analysis tools, especially those that perform interprocedural pointer analysis, can detect many errors in STL usage that result in the use of invalid pointers, such as writing beyond the end of the memory allocated for a vector or attempting to use an iterator in an incorrect way. We claim, however, that the existing tools for static analysis are not powerful enough to detect some of the classes of most common errors made by STL users and, conversely, that their diagnostics often are too weak to be helpful to users. The idea of an STL-aware static analysis therefore is to view STL components at their natural level of abstraction in the library, i.e., at the level of their specification, and to perform checking based on the behavior at the STL component level instead of at the language primitive level. As we will show, this high-level view allows both detecting more types of errors and giving better diagnostics that directly use the terminology of STL. Furthermore, the high-level view reduces the complexity of the analysis.

Figure 3 compares the programmer's view with the view of a traditional compiler. From the "high-level" view of an STL component, as taken by the programmer, the loop listed in this figure consists of a sequence of operations that add an element to the end of the vector, increasing its size and perhaps causing it to allocate more memory. The "low-level" view of the STL vector, as seen by the compiler, looks very different: after inlining, the call to `push_back` has been replaced by a sequence of operations involving pointer arithmetic, a subroutine call, and allocations and deallocations within the loop. In contrast to this traditional approach of inlining, the STL-aware static analysis takes a view of the example code that closely resembles the programmer's view and replaces the call to `push_back` by a semantic specification for `push_back` that denotes the behavior of a routine at its conceptual level and without regard to the underlying implementation. For example, the specification for `push_back` increments the size of the vector and updates internal information about reallocations and iterator invalidation.

What information exactly gets lost in the traditional inlining step? To explain the limitations of a "low-level" view we look at a common vector implementation as depicted in Figure 4, consisting of the three pointers `start`, `finish`, and `end-of-storage`. The `start` pointer references the first element in the container, the `finish` pointer points to one memory cell beyond the last element in the container, while the final pointer, `end-of-storage`, refers to one memory cell beyond the last memory cell that has been allocated. Thus, the *size* of the container (i.e., number of elements) is the distance between the `start` and `finish` pointers and the *capacity* is the distance between the `start` and the `end-of-storage` pointers. As a vector expands with `push_back`, the `finish` pointer will be moved toward the `end-of-storage` pointer until both pointers are

<pre>vector<int> v;
for (int i = 0; i < N; ++i)
{
 v.push_back(i*i);
}</pre>
	<pre>vector<int> v;
	for (int i = 0; i < N; ++i) {
	  if (v.finish == v.end_of_storage) {
	    int size = v.finish-v.start;
	    int new_size = size? size*2 : 1;
	    int* new_start = new int [new_size];
	    copy(v.start, v.finish, new_start);
	    delete [] v.start;
	    v.start = new_start;
	    v.finish = v.start+size;
	    v.end_of_storage = v.start+new_size;
	  }
	  *v.finish++ = i*i;
	}</pre>

Figure 3. A simple loop that uses an STL vector (left) and the partially inlined version of the same loop based on a common vector implementation scheme (right).

equal, where the size of the vector has now reached the capacity of the vector, and later insertions will require the allocation of more memory to continue. Such an implementation is assumed in the expanded code of Figure 3.

Given the model vector implementation implied by Figure 4, we can now explain why traditional static analysis is unable to detect the STL usage errors that Figure 1 illustrates, where the code incorrectly attempts to write values directly to the end of a vector. The end of the vector corresponds to the `finish` pointer and we can see that there may be allocated memory between the `finish` and `end-of-storage` pointers. From the low-level pointer-centric view of an STL vector implementation there therefore is no error in writing beyond the end of a vector—unless the size of the vector `values` exceeds the allocated capacity of the `results` vector and the loop runs beyond the end of allocated storage, a traditional static analysis does not diagnose Figure 1 as incorrect code. Only an analysis that is aware of the vector specification and the fact that it is illegal to dereference the end iterator of a vector, will be able to diagnose the error in Figure 1.

An STL-aware analysis is also able to analyze code in a platform-independent manner. Though we have given one sketch of an implementation of a STL vector, there are certainly other ways to construct an

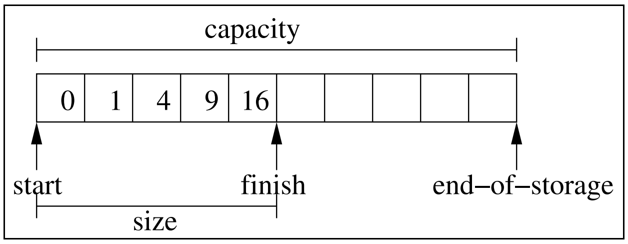

Figure 4. A model of a typical STL vector implementation

STL vector; with more complex STL data structures, such as a `multiset` or `map`, implementations may vary considerably yet still meet the behavioral requirements outlined in the C++ standard. An STL-aware static analysis uses a specification of the behavior required by the C++ standard and thus, by construction, covers all correct implementations, so that checking a program on a single STL platform suffices to ensure correct use of STL on other STL platforms. A traditional static analysis cannot have this ability, because it is impossible to enumerate all possible implementations to test against.

Using specifications as the basis for the analysis of STL components additionally reduces the complexity of the analysis. Already the replacement of the simple vector class—typically containing three pointers—with its specification—requiring only two integers—makes the analysis problem simpler, because integer arithmetic is vastly simpler to analyze than pointer arithmetic, and the integers used in the vector specification are grouped into a single, cohesive component. Within a traditional analysis, the three pointers that comprise a typical vector cannot be coalesced into a single entity and their logical relations to the associated vector as well as to each other is lost. For more complicated data structures, such as the red-black trees often used to implement the STL `map`, not even shape analysis [6, 12] can give the compiler insight into the workings of the data structure.

One final reason remains for the choice of an STL-aware analyzer instead of a traditional analyzer. Much as syntactic error messages with the use of STL are largely indecipherable and are quite detached from the actual code written by the user, the error messages produced by a traditional analyzer references pointers and operations that lie deep within the STL implementation, with very little obvious correlation to the user's task. Indeed the notions of containers and iterators do not exist at the pointer level, but are abstract concepts present in the STL. With an STL-aware analysis, error messages can describe problems in the terminology of the STL, and in a manner that is independent of the underlying implementation.

In summary, an STL-aware analysis has the potential to be more effective for STL programs than a traditional analysis: it can diagnose more problems related to high-level behavior, which is not modeled in the low-level implementation, can detect them more efficiently and can give better feedback to the user when a problem is found. In the remaining sections we present and discuss our implementation of an STL-aware static analyzer.

3. Examples

For a first impression of the new STL-aware analyzer we demonstrate its capabilities with three examples. Since it is one of the most common STL usage errors to misuse the end iterator of an STL vector as an intended insertion pointer, our first example is again Figure 1. As the following output shows, the STL-aware static analysis correctly detects the error:

```
"/home/gregod/Projects/SGI/sgi/edg/libcomo/stl_algo.h", line 583: error:
          attempt to dereference a past-the-end iterator
      *__result = __opr(*__first);
      ^

  in call to function transform at meyers.cpp, line 13

"/home/gregod/Projects/SGI/sgi/edg/libcomo/stl_algo.h", line 582: error:
          attempt to increment a past-the-end iterator
    for ( ; __first != __last; ++__first, ++__result)
                                          ^

  in call to function transform at meyers.cpp, line 13

2 errors detected in the compilation of "meyers.cpp".
```

Indeed the analyzer produces two diagnostics: the first is the error we expected, stating that we are attempting to dereference a past-the-end iterator (the end iterator of the `results` vector). The second diagnostic is perhaps unexpected, but nonetheless correct, and related to the increment operation that advances an iterator after a value has been written through it. Correctly, the static analysis diagnoses an error when the code attempts to increment a vector iterator beyond the end of the vector. As described in Section 2, the first error may or may not be diagnosed by a traditional static analysis, and the second error would not be diagnosed. A run-time check, such as that done by STLport, would have reported an error similar to the first error given by our analysis.

The situation is different in the second example in Figure 5, which illustrates an attempt to find the first employee named “Bjarne” within a list of employees. If no employee named “Bjarne” is found the return

```
vector<Employee> employees; // fill employees
Employee& bjarne =
  *find_if(employees.begin(), employees.end(),
           HasFirstName("Bjarne"));
```

Figure 5. The call to `find_if` may return a past-the-end iterator that must not be dereferenced.

value of `find_if` will be `employees.end()`, which must not be dereferenced because it is a past-the-end iterator. A traditional analysis might not catch such an error because the iterator may not refer to unallocated memory, and a run-time analysis catches such an error only if the list of employees did not contain a "Bjarne." Our static analyzer, however, detects the error independently from a particular input:

```
"find.cpp", line 41: error: attempt to dereference a past-the-end iterator
      *std::find_if(employees.begin(), employees.end(),
      ^

1 error detected in the compilation of "find.cpp".
```

Finally, we revisit the sorted vector insertion in Figure 2 that illustrates a case of potential iterator invalidation. Again, run-time analysis such as the one in STLport can only determine this example to be incorrect under very specific circumstances that vary with the STL platform, whereas our static analyzer emits the following error messages at compile time:

```
"sorted_vec_insert.cpp", line 11: error: attempt to copy a singular
          iterator
    return lb;
    ^
  in call to function insert_in_sorted_vector at sorted_vec_insert.cpp,
          line 23

"sorted_vec_insert.cpp", line 24: error: attempt to copy a singular
          iterator
    v.insert(pos, 41);
             ^

"sorted_vec_insert.cpp", line 24: error: attempt to insert into vector
          using a singular iterator
    v.insert(pos, 41);
    ^

3 errors detected in the compilation of "sorted_vec_insert.cpp".
```

Here the analyzer has determined that the insertion within `insert_in_sorted_vector` might cause reallocation, which then makes the iterator `pos` that was created prior to the insertion singular. This singular iterator is copied twice (an illegal operation, see [3]), and then used in another assertion. Our static analyzer emits error messages for each of these incorrect operations.

4. Implementation

The implementation of our static analyzer consists of approximately six thousand lines of C++ code and has been integrated into the Edison C++ front end [2]. It relies on several forms of static analysis, including a simplistic pointer analysis and more sophisticated analyses of integers. Since most STL algorithms iterate ("loop") over ranges, the identification of induction variables in loops and symbolic loop trip count derivations form the current core of the analyzer.

We developed a small language named Semple that is used to specify the behavior of components. It is a simple language in that it only supports a limited set of data types and only the most basic control-flow constructs, but it also includes assertions that are used to check correctness. Associated with each assertion is a simple text string that denotes the error message to be reported to the user. The error messages given in Section 3 are in fact just text strings within the Semple specification for the STL vector and vector iterator types. Figure 6 illustrates the Semple specification for a vector iterator and its dereferencing operation (`operator*` in C++). The assertions check that all conditions required of a dereferenceable iterator are satisfied: the iterator must have an associated vector, and its position must be within the valid bounds of the vector. Additionally, the iterator version must match the version of the vector. Vector iterators are given the version number of the vector that creates them (e.g., via `vector::begin`); if reallocation is performed by the vector, its version number is incremented, thus invalidating all existing iterators. The author of the specification has also given specific strings that are used in error messages (see examples in Section 3). We expect that Semple specifications will be written not by average users, but by the designers of components and libraries. We also stress that the Semple specifications, hence the analysis, is in fact not restricted to the STL, but may be used for the semantic specification of any component.

Analysis proceeds first by compiling C++ code into the equivalent Semple code, replacing STL components and operations on those components with the Semple specifications. The mapping from STL components and functions to their Semple equivalents is performed in an ad hoc way at

```
class std_vector_iterator {
  vector :  ↑std_vector; // associated vector
  pos :  integer;       // position of the iterator
  version :  integer;   // track iterator invalidation
}
function std_vector_iterator_dereference(
   i :  ↑std_vector_iterator) :  ignore
{
  assert(i→vector != nil and
         i→version == i->vector->version,
         "attempt to dereference a singular iterator");
  assert(i→pos ≥ 0 and i→pos < i→vector→size,
         "attempt to dereference a past-the-end iterator");
}
```

Figure 6. The Semple specification for a vector iterator and its dereference operation

present, using custom pragmas strewn throughout an implementation of STL.

We then interpret the Semple representation of the program, using symbolic placeholders to denote the values of program variables that cannot be known at compile time. At program points that are reachable via several paths of execution, the symbolic values of program variables may be different depending on the specific executation path taken to that program point. We rectify these multiple incoming symbolic values for a single program variable by assigning that program variable a symbolic value that is restricted to a range encapsulating all incoming values. Conditional branch tests are evaluated using the range test [5] that determines the truth values of integer relationships given symbolic bounds on the program variables involved. Interpretation of entire programs thus requires a large amount of symbolic computation, for which we have chosen the GiNaC [4] library.

For loop analysis we use the symbolic differencing method due to Haghighat and Polychronopoulos [7]. In symbolic differencing, in short, a loop is executed symbolically some number of times. After each iteration Newton's forward formula for interpolation is applied, to determine a function for each integral expression dependent only on the loop iteration number and the initial conditions. Symbolic differencing can find closed form induction expressions of arbitrary degree m by executing the loop $m + 2$ times, although in STL code one rarely finds induction expressions of degree higher than one. To determine the actual effects of a loop on program variables, we simulate the execution of all iterations of the loop in a single pass and use the results of induction variable recognition to compute the final values of program variables.

Figure 5 illustrates where loop analysis is required for the checking of STL usage. The example code invokes `find_if`, whose internal loop can run from zero to n times, where n is the distance between the beginning and ending iterators given to the algorithm. Therefore, the iterator returned from the algorithm may be anywhere between the beginning and ending iterators, inclusive. Since the ending iterator must never be dereferenced, the second assertion for iterator dereferencing in the Semple specification (given in Figure 6) fails to be true throughout the range of return values from `find_if` and ultimately triggers an error message.

The final analysis implemented in our static analyzer is a simplistic pointer analysis. Pointer analysis is required when iterators are copied and created, because the specification of the vector iterator (see Figure 6) keeps a pointer to the vector it references so that its validity can be checked. Pointer analysis is also necessary because C++ pointers and references are mapped to pointers in Semple, and user pointers must also be analyzed. Pointer analyses are often the least scalable analyses to larger programs, so we cannot assess the scalability of our approach until a more complete pointer analysis has been implemented.

5. Related and Future Work

The Canvas project [1] seeks to allow component designers to specify component conformance constraints and then verifies clients of those components using the conformance constraints. This approach is similar to our own, using an external specification language and static analysis to check for proper component use. Also similiar to our approach, the Canvas project has focused on containers and iterators, although for the Java language and libraries. From an implementation point of view, we have used primarily an integer static analysis, whereas the Canvas project has chosen to focus on pointer analysis and the derivation of specialized heap analyses to boost efficiency [11].

Our work can be viewed as a step toward enabling Active Libraries [15]. Active libraries are libraries that take an active role in their own compilation, e.g., by tuning themselves to generate better code or by integrating with development tools so that the library is viewed from the user's perspective and not the implementor's perspective. Our static analy-

```
template<typename InputIterator, typename F>
F for_each(InputIterator first, InputIterator last, F f)
{
    for( ; first < last; ++first)
      f(*first);
    return f;
}
```

Figure 7. This implementation of `for_each` is incorrect, because input iterators are not required to support `operator<`.

sis allows libraries to take an active role in semantic analysis during compilation.

Concept checking in C++ [13] has introduced the notion of *concept archetypes*, which are abstract data types that supply only the basic syntactic behavior of the concepts they model. For instance, an archetype for the Input Iterator concept supports only the basic operations allowed on an input iterator. Concept archetypes are used in the syntactic checking of generic algorithms against the concepts they are stated to require. Figure 7 illustrates an incorrect implementation of the STL `for_each` algorithm, where the implementor has used the `<` operator to compare iterators. While this algorithm compiles and works properly for any random access iterator, such as that of a STL vector or deque, it will fail to compile when given a true input iterator, because input iterators are not required to support the `<` operation. Instantiating this invalid `for_each` implementation with an input iterator archetype will immediately reveal the error, because the archetype will not define a `<` operation.

Concept archetypes themselves cannot check for well-defined runtime behavior, because they contain no semantic information. However, one can view a concept archetype merely as a component, and the generic algorithm as a client of that component; then the behavior of the concept archetype can be specified within the Semple language and our static analysis can be applied to the algorithm. We hope to integrate Semple specifications with the concept specifications as input to Caramel [16], a project that seeks to unify the generation of concept documentation, checking code, and archetypes for C++. An extension to this system would enable Semple specifications to be emitted with concept archetypes so that both syntactic and semantic information is present in a single document and may be verified with one analysis run.

In the future we will extend the static analyzer in three directions. First, we will replace the simplistic pointer analysis with a more powerful form of pointer and alias analysis, allowing for more complex relationships to be checked. Second, we will extend the analyzer to include recursive routines. The whole class of sorting routines, along with other divide-and-conquer routines, will then become accessible to static analysis. Third, we will open the analysis so that it is able to handle arbitrary user data types. STL algorithms and components then can be checked even when used with user-defined adaptors.

In addition to extending the capabilities of the static analysis, we wish to extend the capabilities of authors of Semple specifications. While the use of static strings for error messages produces reasonable results, we can get better results by giving the specification writer more power over the formatting of error messages. For instance, an error message for the example given in Figure 2 could reference the operation causing the invalidation.

Checking of code can often turn to optimization of the same code. The example code in Figure 3 could execute faster if it were known that the branch performing reallocation would never be taken. Since the static analysis derives this information to determine whether iterator invalidation will occur, it can pass this information on to later optimization stages. Furthermore, an STL-aware optimizer could force the branch never to be taken by using the STL vector's `reserve` function to ensure that the capacity of the vector will never be exceeded within the loop.

Acknowledgments

This work was supported in part by the National Science Foundation (NSF) NGS Grant 0131354 and by Silicon Graphics, Inc., California. We like to thank the Edison Design Group [2] for making their C++ front end available to us. Thanks to the EDG code, its design and the extensive documentation, we were able to make progress much faster than expected. David Musser suggested to distinguish between singular and past-the-end iterators.

References

[1] The Canvas project. `http://www.research.ibm.com/menage/canvas/`.

[2] Edison Design Group C++ front end. `http://www.edg.com/`.

[3] ANSI-ISO-IEC. *C++ Standard, ISO/IEC 14882:1998*, ANSI standards for information technology edition, 1998.

[4] C. Bauer, A. Frink, and R. Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *J. Symbolic Computation*, (33):1–12, 2002.

[5] W. Blume and R. Eigenmann. Symbolic range propagation. In *Proc. of the 9th Inter. Parallel Processing Symposium*, pages 357–363, April 1995.

[6] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL*, pages 1–15. ACM, January 1996.

[7] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996.

[8] B. McNamara and Y. Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming*, October 2000.

[9] S. Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley, 2001.

[10] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library*. Addison Wesley, 2nd edition, 2001.

[11] G. Ramalingam, A. Warshavsky, J. H. Field, and M. Sagiv. Deriving specialized heap analyses for verifying component-client conformance. Research Report RC22145, IBM Research Division, August 2001.

[12] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Symposium on Principles of Programming Languages*, pages 16–31. Association for Computing Machinery, January 1996.

[13] J. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, October 2000.

[14] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-95-11, Hewlett Packard, November 1995.

[15] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

[16] J. Willcock, J. Siek, and A. Lumsdaine. Caramel: A concept representation system for generic programming. In *Second Workshop on C++ Template Programming*, October 2001.

[17] L. Zolman. An STL error message decryptor for Visual C++. *C/C++ User's Journal*, July 2001.