SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2005; **1**:1 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

STLlint: Lifting static checking from languages to libraries

Douglas Gregor[1] and Sibylle Schupp[2]

[1] *Computer Science Department*
Indiana University
Bloomington, IN 47405 USA
⟨`dgregor@cs.indiana.edu`⟩
[2] *Dept. of Computing Science*
Chalmers University of Technology
SE-41296 Gothenburg
⟨`schupp@cs.chalmers.se`⟩

SUMMARY

Traditional static checking centers around finding bugs in programs by isolating cases where the language has been used incorrectly. These language-based checkers do not understand the semantics of software libraries, and therefore cannot be used to detect errors in the use of libraries. In this paper, we introduce STLlint, a program analysis we have implemented for the C++ Standard Template Library and similar, generic software libraries, and we present the general approach that underlies STLlint. We show that static checking of library semantics differs greatly from checking of language semantics, requiring new representations of program behavior and new algorithms. Major challenges include checking the use of generic algorithms, loop analysis for interfaces, and organizing behavioral specifications for extensibility.

KEY WORDS: static checking, program verification, interface compilation, generic programming, Standard Template Library

1. Introduction

Static program checkers attempt to locate incorrect constructs in programs and report them to the user before the programs are actually executed. Each static checker isolates problems for a different set of program constructs, with many checkers, such as the venerable `lint` [1] checker for the C language, detecting some set of common programmer errors in the use of the language, such as non-portable type casting and unused variable definitions. More advanced tools [2, 3, 4] detect errors such as NULL pointer dereferences or out-of-bounds array accesses. Each of these `lint`-like static checkers operate

Contract/grant sponsor: National Science Foundation (NSF); contract/grant number: 0131354

Received 21 January 2004
Revised 9 February, 2005
Accepted 17 February, 2005

```
template<class RandomAccessIterator>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

1. **Effects:** Sorts elements in the heap `[first, last)`.
2. **Complexity:** At most $N \log N$ comparisons (where `N == last - first`).
3. **Notes:** Not stable.

Figure 1. Specification of the `sort_heap` algorithm excerpted from the C++ standard [6, §25.3.6.4].

at the level of abstraction of the language, checking programs against the semantics of the underlying language.

Modern mainstream programming languages such as Java [5] and C++ [6] boast large standard libraries of reusable algorithms and data structures that offer functionality beyond what is intrinsic to the language. The use of these and other software libraries represents a shift from reliance entirely on language semantics to reliance on both language and library semantics. Since static checkers are primarily a programmer aid, we claim that programmers using software libraries would benefit from static checkers that check the uses of the software libraries, not just the underlying language. Checking against software libraries is sometimes called client-conformance checking [7] or interface compilation [8].

Fig. 1 is an excerpt of the specification for the `sort_heap` algorithm in the C++ standard [6], originally part of the C++ Standard Template Library (STL) [9, 10]. This specification illustrates how the problem of checking library usage is markedly different from that of checking language usage. The *effects* clause describes several preconditions and postconditions, including:

- The parameters `first` and `last` are iterators [11] that must form a *valid range*. Iterators abstract the notion of iterating over a sequence of values, such as values within a container. Iterators form a valid range `[first, last)` when one may access the elements referenced by `first`, `++first`, `++(++first)`, etc., up to (but not including) the element referenced by `last`.
- The values in the sequence referenced by `[first, last)` must be arranged as a heap. It is specified elsewhere in the C++ standard that the function (or function object) `comp` will provide the ordering relation for the heap if it is supplied; otherwise the $<$ operator will provide the ordering relation.
- This algorithm will rearrange the elements referenced by `[first, last)` into sorted order, based on the ordering relation `comp` or the $<$ operator.

We note that these pre- and post-conditions are very high-level semantic properties. The notion of a valid range is based on the iterator *concept* [12, 13], i.e., a set of syntactic and semantic requirements coupled with a set of abstractions that fulfill these requirements. Thus to check that the parameters `first` and `last` form a valid range, we aren't checking requirements based on language constructs (the C++ *language* does not have iterators), or even based on a concrete type or interface, but are checking against requirements specified for an abstract data type. Moreover, the "heap" precondition and "sorted" postcondition describe particular orderings on the values that will be referenced by the given iterators. Overall, static checking given such a high-level specification must cope with abstractions far from the language itself, including user-defined data types described only by their conceptual requirements, and must provide checking for semantic guarantees written in terms of these abstractions.

We present here the challenges of "lifting" a static checker from the language level to the level of abstract, generic software libraries. We focus on the widely-used C++ Standard Template Library (STL) and its abstractions, and present our "higher-level" static checker STLlint [14] that performs checking of the use of the STL. Characteristic for our approach is that it is based on symbolic execution rather than on abstract interpretation, which proved to be too imprecise for our purpose. While most of the symbolic techniques we employ are not new themselves, we combine them in a way that they together can address the unique challenges of library analysis. In addition, we present the first attempt at user-centric, extensible static checking for generic software libraries such as the C++ STL. The resulting tool, STLlint, can handle STL in its entirety, with no restrictions; empirical tests of STLlint include comprehensive sets of examples from two standard textbooks and showed a low false positive rate of 0.59%.

This article is presented as follows: Sec. 2 describes the motivation behind STLlint, illustrating the shortcomings of language-level static checking and the need for library-aware static checkers. We then describe the challenges we have faced in the construction of STLlint and the methods we have used to overcome those challenges, including our approach to checking with abstractions in Sec. 3, the challenges of dealing with iterators as higher-level induction variables in Sec. 4, and a way to manage extensibility in a large generic library in Sec. 5.

2. STLlint

STLlint is a static program checker that checks the use of C++ libraries, in particular the STL. Thus instead of treating C++ libraries in the same manner as the user's C++ code, STLlint treats libraries as extensions to the language itself, so that it may check user code against the semantics of the library and not the semantics of the implementation within the language, allowing STLlint to find errors such as:

- Use of an iterator before it has been defined to reference a value in a sequence
- Use of an iterator that has been invalidated by a previous operation, e.g., erasing the element it references
- `vector` or `deque` subscript out of (logical) bounds
- Attempt to dereference a past-the-end iterator
- Attempt to apply a binary search to a sequence that has not been sorted, or has been sorted based on a different ordering relation

A powerful language-based, `lint`-like tool could be used to find many errors that STLlint can find, by analyzing the user program along with the source code for the implementation of the library, since most—but not all—errors in the use of a library result in errors in the use of the language. For

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

Figure 2. Language-level tools cannot detect the erroneous attempt to dereference a past-the-end iterator in this example, because the language semantics are not violated.

instance, an attempt to access the value referenced by a `list` iterator after that element has been erased may result in a warning involving the use of memory that has already been freed. However, typical library-level optimizations such as memory pooling may mask such errors even from run-time memory checkers.

Furthermore, there are errors in the use of libraries that no `lint`-like checker is able to detect because they do not result in errors in the use of the language. The example in Fig. 2, adapted from Meyers' item #30 [15], illustrates one such case. Here, we are attempting to apply the `transform` operation that, for each value `x` in the `values` vector, writes the result of `transmogrify(x)` into the corresponding position in the `results` vector. However, the `results` vector is empty, so the `transform` operation attempts to dereference a past-the-end iterator (`results.begin()`) when it writes the result, invoking behavior undefined by the C++ language. STLlint will diagnose this error, but a `lint`-like tool cannot: the `reserve` call prior to the `transform` operation allocates a suitable amount of memory, such that within the `transform` call, the program always accesses memory that has already been properly allocated. In essence, the program is not an error from the perspective of the C++ language but is instead an error from the (more abstract) library perspective. By considering only the implementation of the C++ library and not its higher-level semantics, `lint`-like tools cannot detect many errors important for the proper use of libraries.

The program in Fig. 2 can be trivially modified to use data structures much more complicated than vectors, such as linked lists, double-ended queues, or even balanced binary trees. While the user code is not complicated by these changes due to the iterator abstraction provided by the STL, the underlying implementation becomes drastically more complex, requiring a `lint`-like tool to perform more advanced (and expensive) analyses, such as shape analysis [16, 17] to verify proper traversal of a linked list. STLlint, on the other hand, treats all of these cases similarly by analyzing the abstraction, not the implementation. In essence, STLlint applies abstractions for the same reason that a programmer would: to reduce the complexity of a task by factoring out subtasks. Since these abstractions are not language abstractions, a language-level static checker could not do the same.

Errors that a `lint`-like checker can detect will be reported where the C++ language semantics have been violated, often deep within the implementation of the STL and far from the user code that is in fact incorrect. Being unable to relate the library implementation back to its interface, `lint`-like tools cannot provide the user with diagnostics that relate to the user's error. A similar problem occurs with the type checkers of C++ compilers, where errors in the use of C++ template libraries are reported within the library implementations instead of at the point of use [18, 19, 20].

STLlint is comprised of three modules: a static analysis engine, a C++ language interface, and a description of the behavior of the C++ standard library suitable for analysis. The static analyzer verifies assertions within the program representation, emitting diagnostics when the assertions cannot be proven true. The C++ language interface parses C++ source code (using the Edison Design Group's C++ front end [21]), translates the C++ representation into our Semple internal representation language (used by the static analysis engine), and finally translates diagnostics produced by the static analyzer into error messages that integrate with the C++ front end. Finally, STLlint contains a description of the semantics of the C++ standard library (further discussed in Sections 3 and 5), written in C++ and integrated with a derivative of the SGI Standard Template Library implementation [22]. This semantic description is not compiled into the STLlint binary, but is parsed by the C++ compiler when it is needed.

The rigid separation of the language interface and library semantics from the static analysis engine of STLlint ensures that the analysis can be applied to other languages and other libraries. While STLlint does include a semantic description of the C++ standard library, this description is in no way special: other C++ libraries may use or extend that description, or may provide a completely different description to be verified by the static analysis engine.

3. Static checking with abstractions

STLlint employs executable specifications, written within the source language (C++), to provide the analyzable abstractions used in analysis. With STLlint, the program code is parsed in its entirety and the types and functions that have specifications are directly replaced by the specifications themselves. The entire program (with specifications) is then analyzed by the static checker, and any assertions within the program or specifications are verified. We do not attempt to verify the accuracy of implementations with respect to the specifications they must meet [23, 24], although other static checkers have demonstrated that this capability is useful [3], because verification of generic algorithms [25] is beyond the scope of STLlint.

3.1. Executable specifications

Executable specifications in STLlint are written within a statically type-safe subset of C++. The subset includes templates, classes, functions and methods, (multiple) inheritance, pointers, integers, and most control structures, but does not include unions, type casts, floating-point arithmetic, or exceptions. Additionally, STLlint specifications may contain assertions (i.e., conditions on program variables that must hold true; see Fig. 3), assumptions (i.e., conditions that should be assumed true even if they cannot be proven), and two features that cannot be directly expressed in C++: a `foreach` loop that iterates over the entire program state space and a set of classification statements that allow C++ objects to be augmented with additional information at run time (of the analyzer).

The `foreach` construct allows one to iterate over all objects of a given type within the (abstract) program state as represented by the static analyzer. This construct allows first-order logic to be

```
iterator erase(iterator pos) {
  semple_assert(pos.dereferenceable(),
    "attempt to erase a singular or past-the-end iterator");
  semple_assert(pos.sequence_ == this,
    "attempt to erase an iterator from another container");
  semple_foreach(iterator i) {
    if (i.sequence_ == this && i.position_ >= pos.position_)
      i.version_ = 0;
  }
  --size_;
  return iterator(pos.position_, this);
}
```

Figure 3. Specification of the `erase` operator for an STL `vector`.

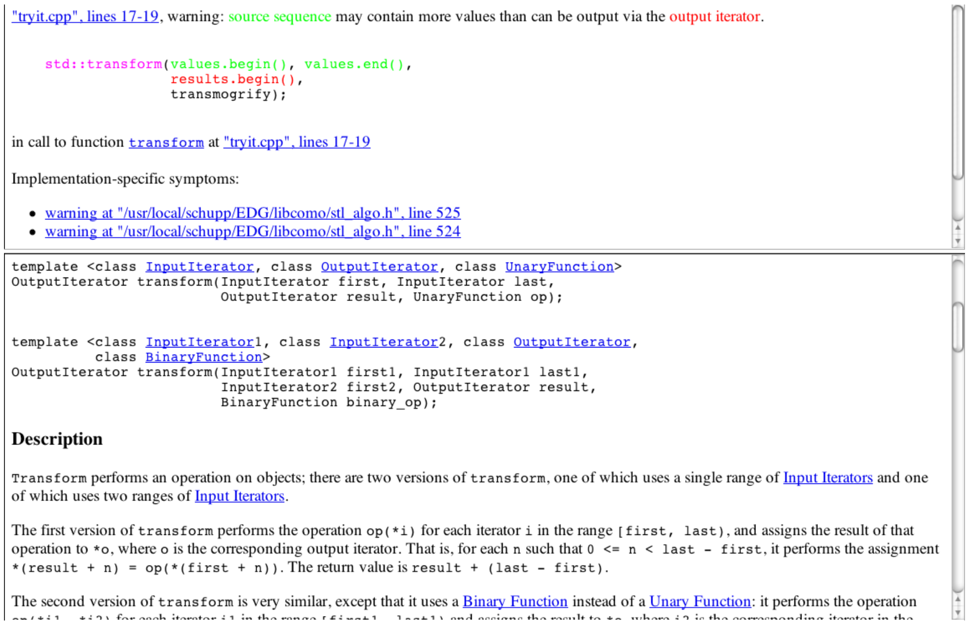
employed by STLlint specifications, which is required when object sharing is involved. For instance, many iterators may share the same container object and operations that modify that container need also modify the associated iterators. Fig. 3 illustrates the specification of the `erase` operation of an STL `vector`, which employs the `foreach` construct to invalidate all iterators that reference the erased element or any element following it [6, §23.2.4.3].

The classification constructs of STLlint allow specifications to attach additional analysis-only data to any object representation. The feature allows, for instance, the specification of a sequence-sorting routine (such as `sort_heap`, from Fig. 1) to attach a tag to the sequence indicating that the sequence has been sorted, which may later be queried by other specifications (e.g., a binary search) or removed (e.g., by a sequence randomization algorithm). Any number of classifications may be applied to a particular object, and classifications are themselves objects that may have state. Classifications may be queried (via an "is-a" check), accessed as an object (via the "as-a" operation), and eliminated (via the "declassify" operation). The usefulness of classifications, and especially their use as a specification extension mechanism, will be covered in Sec. 5.

3.2. Preserving library/user code separation

Specifications provide the means to hide implementation details from the static checker, to reduce the complexity of the analysis, but we must also perform the inverse operation by hiding the specification details from the end user. For instance, if we are performing static checking at the abstraction level of iterators and containers, we should produce diagnostics that refer to iterators and containers, not the pointers or classes that are used to implement them. Failure to phrase error messages in terms of the abstractions the user has applied eliminates the primary end-user benefits of a higher-level static analyzer.

Producing diagnostics at the same level of abstraction as the program code requires domain-specific knowledge of the abstractions. This knowledge may be built into the static checker (as are the semantics of a programming language in a `lint`-like checker) or included with the specifications themselves. STLlint employs the latter technique, where all assertions (checks) within the specification



"tryit.cpp", lines 17-19, warning: source sequence may contain more values than can be output via the output iterator.

```
    std::transform(values.begin(), values.end(),
                   results.begin(),
                   transmogrify);
```

in call to function `transform` at "tryit.cpp", lines 17-19

Implementation-specific symptoms:

- warning at "/usr/local/schupp/EDG/libcomo/stl_algo.h", line 525
- warning at "/usr/local/schupp/EDG/libcomo/stl_algo.h", line 524

```
template <class InputIterator, class OutputIterator, class UnaryFunction>
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryFunction op);


template <class InputIterator1, class InputIterator2, class OutputIterator,
          class BinaryFunction>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, OutputIterator result,
                         BinaryFunction binary_op);
```

Description

`Transform` performs an operation on objects; there are two versions of `transform`, one of which uses a single range of Input Iterators and one of which uses two ranges of Input Iterators.

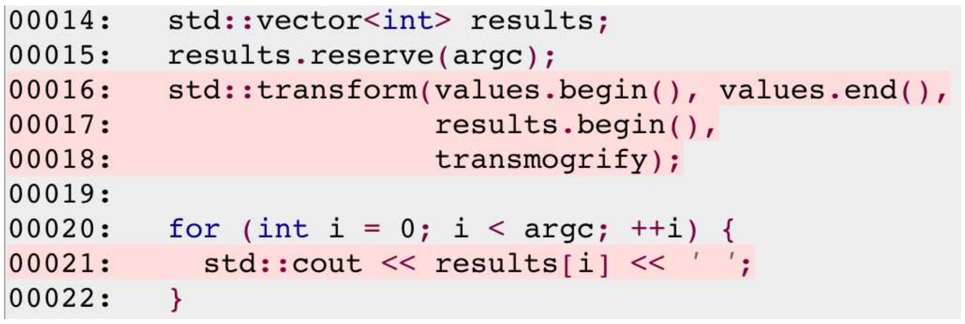
The first version of `transform` performs the operation `op(*i)` for each iterator `i` in the range `[first, last)`, and assigns the result of that operation to `*o`, where `o` is the corresponding output iterator. That is, for each `n` such that `0 <= n < last - first`, it performs the assignment `*(result + n) = op(*(first + n))`. The return value is `result + (last - first)`.

The second version of `transform` is very similar, except that it uses a Binary Function instead of a Unary Function: it performs the operation `op(*i1, *i2)` for each iterator `i1` in the range `[first1, last1)` and assigns the result to `*o`, where `i2` is the corresponding iterator in the

Figure 4. HTML output produced by STLlint when it is invoked on the program given in Fig. 2.

are accompanied by a diagnostic to be emitted should the assertion fail to prove true. Error messages are typically placed on entry to the library and refer to the context of the caller, using simple mark up commands to highlight function call arguments of interest to the user.

Additionally, function specifications provide hypertext links to external documentation that may be presented to the user along with the diagnostic message identifying the source of the error. This way, STLlint provides both domain-specific information at the point where the user error occurred along with documentation to help the user understand the semantics of the function under question. Fig. 4 illustrates the HTML output produced by STLlint for the example in Fig. 2, including argument highlighting and documentation for the `transform` function (derived from the SGI STL documentation [22]). In addition, STLlint groups diagnostics triggered from within an executable specification under any diagnostics generated from the specification's preconditions (under the heading "Implementation-specific symptoms" in Fig. 4). The effect is to present the user first with a diagnostic relating only to the potentially-incorrect user code (at the algorithm's level of abstraction), but to also provide links to other diagnostics that allow the user to "step into" the specification/implementation of the algorithm to see how mistakes at the call site affect the underlying algorithm. Finally, STLlint generates hyperlinks from source-code references to stylized versions of the source code that highlight lines in the source code for which STLlint has emitted diagnostics, allowing the user to quickly determine the context of a diagnostic. Fig. 5 illustrates this highlighting behavior on a sample program including Fig. 2.



```
    std::vector<int> results;
    results.reserve(argc);
    std::transform(values.begin(), values.end(),
                   results.begin(),
                   transmogrify);

    for (int i = 0; i < argc; ++i) {
      std::cout << results[i] << ' ';
    }
```

Figure 5. STLlint highlights the lines referenced by its diagnostics (hyperlinked from the diagnostic messages themselves) within the source file, permitting the user to see the errors in context.

4. Higher-level iteration constructs

Perhaps the greatest challenge in lifting STLlint to library-level analysis has revolved around loop analysis, which must now cope with iterators. Unlike iterators in other languages and libraries, such as the Java Collections Framework [26], many STL iterators can be used to reference a single sequence concurrently, making modifications and moving through the sequence in arbitrary ways, resulting in complex iteration patterns. STLlint therefore requires sophisticated loop analysis and particularly induction variable recognition with “higher-level” induction variables. Induction variables are traditionally integer variables that increase by some constant value each loop iteration, but have been extended to more complex induction expressions, multiple assignments, wraparound variables, etc. These forms of induction variables are adequate for low-level internal representations and even some constructs in high-level languages, for instance using integer variables to step through arrays.

Induction variables may also take on other forms, such as pointers with pointer arithmetic or traversal of a linked-list data structure. In the former case, it suffices to represent pointers by a pair $(address, position)$, where $address$ is the address of the beginning of an array and $position$ represents the offset of the pointer into the array, in which case traditional loop analysis techniques can be applied to the $position$ provided the $address$ remains constant throughout the loop. The latter case, however, does not permit such a simple solution, and the situation worsens with more complex data structures.

The iterator abstraction provides the ability to iterate over all elements within a particular container regardless of the underlying data structure, maintaining a consistent interface and semantics regardless of the implementation details. Iterators are typically utilized in program loops, requiring the static checker to adequately support iterators within loop analysis in order to properly verify their use. Since we cannot hope to check the implementation of iterators for complex data structures, we introduce specifications for the iterator that are simpler and are amenable to loop analysis. With STLlint, we have chosen to specify iterators by a pair $(sequence, position)$, where $sequence$ is the address of the data structure the iterator references and $position$ is a 0-based offset representing the location of the iterator in the sequence, as is done with pointer arithmetic. In theory, this representation of iterators

```
vector<Student> extract_fails(vector<Student>& students)
{
  vector<Student> fail;
  vector<Student>::iterator iter = students.begin();

  while (iter != students.end()) {
    if (fgrade(*iter)) {
      fail.push_back(*iter);
      iter = students.erase(iter);
    } else
      ++iter;
  }
  return fail;
}
```

Figure 6. A typical iterator loop wherein induction variables are modified within subroutine calls (`++` and `erase`), the loop termination condition is hidden within a subroutine call (`!=`), and the two induction variables `iter` and `students.end()` are monotonic.

allows us to reuse traditional loop analysis techniques to recognize iterators as induction variables based on the induction behavior of the iterator's $position$.

However, loop analysis on iterator specifications is much more complex than analysis on integer variables, or even on pointers with pointer arithmetic. The analysis must cope with multiple assignments to induction variables, monotonic induction variables, induction variables accessed/modified via (multi-level) pointers, and we must implement an interprocedural loop analysis because the iterator operations that dominate loop analysis, such as the operations that advance or compare iterators, are function calls.

4.1. An example

The deceptively simple loop in the subroutine `extract_fails` in Fig. 6, taken from an introductory C++ text [27], illustrates the challenges for the loop analysis:

- Within the loop, the primary induction variable is the iterator `iter`, which steps through the `students` vector in two ways: via the `++` prefix operator, which increments the $position$ of the iterator by one step, and via the `erase` operation, that in effect leaves the $position$ of `iter` constant but shifts all elements after `iter` in the vector back one step, reducing its size. From the two "increment" operations we see that `iter` is a monotonic induction variable, because its $position$ moves either zero or one steps forward in each iteration.
- The prefix operator `++` is a function call that operates on `iter` via a pointer and `erase` is a function call that returns a new value for `iter` that is later copied into `iter` via a copy constructor, which again operates on `iter` via a pointer. We therefore have multiple assignments to the induction variable `iter` that always occur through one level of pointer indirection within

subroutines, requiring both pointer analysis and the ability to perform interprocedural loop analysis.

- The infix `!=` operator is again implemented via a function call operating on a pointer to `iter` and a pointer to the (temporary) result of the call expression `students.end()`. Thus any algorithm attempting to use the loop termination condition, e.g., to calculate a loop trip count or to perform narrowing operations [28, 29] must be interprocedural and able to cope with pointers and temporary variables.
- Although the expression `students.end()` appears to be loop-invariant, it is not. The "end" iterator of a container is called a "past-the-end" iterator, because it points one element beyond the last element in the container. In our $(sequence, position)$ representation of iterators, this corresponds to the case where $position$ is equivalent to the size of the $sequence$. However, the `erase` operation on the `students` vector eliminates one element and decreases the size of the sequence. When `students.end()` is executed next, its position will therefore be one less than the prior position if `erase` has been executed, or equivalent to the prior position if `erase` has not been executed, making the result of `students.end()` a monotonically nonincreasing induction variable. This again complicates computations based on the loop termination condition, as we now have a monotonically nondecreasing induction variable compared via inequality (`!=`) against a monotonically nonincreasing induction variable, requiring us to determine if the two induction variables may ever become equivalent (terminating the loop) or if they may pass each other because both move in the same iteration.
- Finally, STLlint performs many correctness checks within this loop that are not evident from the loop itself. Of particular interest are the checks performed by the specification of the prefix `*` operator, which returns the element that the iterator references. This operator requires that its iterator argument be "dereferenceable", i.e., the following two conditions must hold:

  1. *The iterator is not singular*: A singular iterator is one that has not been initialized to reference a sequence, references a sequence that no longer exists, or has been invalidated by a sequence operation such as `erase`.
  2. *The iterator is not past-the-end*: A past-the-end iterator is one that is not singular and points one element beyond the last element in the sequence. Past-the-end iterators may come from the `end` member function of a container, but may also be reached by incrementing an iterator referencing the last element in the container.

  Within our representation, a dereferenceable iterator is one that has a valid $sequence$ and a nonnegative $position$ that is strictly less than the size of that sequence. This assertion occurs within the dereference operator `*`, operating on a pointer to an iterator. Since an iterator's $sequence$ is itself a pointer, accessing the size of the sequence from within the dereference operator requires a multi-level pointer access that, in the example from Fig. 6, brings us back to the monotonically nonincreasing size of the `students` vector.

STLlint employs a variety of techniques to analyze iterator loops within a unified *symbolic execution* [30] framework. In the following subsections, we focus on the analysis of integer values, especially as it pertains to the positions of iterators and the sizes of containers, and delve into the enhancements we have made to traditional loop analysis techniques to facilitate higher-level loop analysis with iterators. In preparation, we briefly summarize the core idea of symbolic execution and contrast it with abstract interpretation.

4.2. Symbolic execution

Symbolic execution [30] refers to the process of executing a program given symbolic input in lieu of actual, concrete input. At conditional branches, where different inputs may in fact cause the program to follow different execution paths, the abstract program state is split (“forked”) into two identical copies, one of which assumes that the condition is true and the other assumes that the logical negation of the condition is true. The two program states are modified independently until the paths rejoin, and the program states are merged into a single, conservative approximation of both incoming states.

Symbolic execution is similar to abstract interpretation [29], which also employs a symbolic representation of the values of program variables to perform static analysis. Symbolic execution differs from abstract interpretation in its handling of program loops. While both methodologies perform fixed-point iteration and apply widening/narrowing operations [28, 29] to ensure convergence, symbolic execution employs a different analysis method for program loops that is not dependent on fixed-point iteration.

To better illustrate symbolic execution, we describe the analysis of a single iteration of the loop in Fig. 6; details of the iterator and `vector` specifications involved in the analysis are provided in the appendix. The initialization of `iter` at line 10 sets its position to zero and its sequence to the address of `students`. At the while loop, the iterator’s `operator!=` is invoked, and the analysis maps the actual parameters to the formal parameters of the routine and executes the routine body: assuming that `students` is non-empty, the analysis enters the body of the loop. At the call to `fgrade`, the analysis first steps into the iterator’s `operator*`, which verifies that the iterator is dereferenceable (see specification in Appendix A), and passes the result to `fgrade` (not shown); it is unlikely that we can statically determine the outcome of `fgrade`, so instead we fork the program state into two copies: one assuming that `fgrade` returns true and one assuming that `fgrade` returns false. We then continue the symbolic execution of each branch (separately) to determine that: (1) in the `true` case, the `students` vector decreases in size by one, the `fails` vector increases in size by one, and the iterator `iter`’s position remains constant; (2) in the `false` case, the iterator `iter`’s position increases by one. The two program states are then merged together in a manner that conservatively retains the properties of both states, and the symbolic execution continues with a single abstract program state.

Interprocedural symbolic execution is similar to the call string approach to interprocedural data flow analysis [31] in that it explicitly preserves the call context for every subroutine invocation. For programs involving recursive subroutines, the length of the call string is unbounded, requiring particular care to ensure termination. We therefore treat recursive calls (i.e., those calls to routines already present in the call string) as *widening edges* [28, 29] in the (interprocedural) control-flow graph by applying a widening operation, which is a conservative form of the merge operation, to the program state. STLlint then iterates over the body of a recursive function until the result has stabilized, as is guaranteed by the use of the widening operator [32, §3.3.4].

In an abstract interpretation, the analysis of program loops would proceed similarly to our recursive analysis. However, within symbolic execution we instead apply induction variable recognition, which will be described in Sec. 4.4.1. First, we describe the manner by which the abstract program states are represented, manipulated, and merged.

Throughout the presentation we assume the existence of a context-sensitive, flow-sensitive points-to analysis allowing us to map expressions with pointers to locations in the abstract program state. STLlint employs a simple constant-propagation lattice to describe the values of pointers during the symbolic execution, which is sufficient so long as the sequences associated with an iterator do not differ based on a prior conditional. Our experience has shown that this situation rarely occurs in practice.

4.3. Symbolic integer value analysis

STLlint's static analysis represents the values of integer variables and fields via expressions involving integer literals and symbolic variables. The use of symbolic variables enables STLlint to effectively cope with containers of unknown size and track the relative positions of iterators within a container, which is essential for modeling the invalidation behavior of some STL containers. For instance, the vector `erase` operation (see Fig. 3) invalidates all iterators whose position is greater than or equal to the position of the element being erased, requiring (symbolic) comparisons to determine the set of iterators invalidated by the operation.

Each symbolic variable x falls within a particular value range [33] written $x \in [a : b]$, such that $a \leq x \leq b$. The expressions a and b may again be symbolic, allowing STLlint to represent complex relations amongst integer values. For instance, the position of a particular iterator may be $x \in [0 : N - 1]$ whereas the size of the sequence the iterator references may be N, meaning that the iterator is dereferenceable. Incrementing said iterator results in the position $1 + (x \in [0 : N - 1])$, resulting in an iterator that is either dereferenceable or past-the-end, but is clearly safe to decrement.

STLlint implements the symbolic comparison algorithm discovered by Blume & Eigenmann [34]. This algorithm operates by replacing symbolic variables with their value ranges in a logical order, and then simplifying the resulting expression until it is comparable to zero. For instance, comparing the expression $1 + (x \in [0 : N - 1])$ to N would perform symbolic variable replacements on the difference $1 + (x \in [0 : N - 1]) - N$. Substitution of x for its value range results in $1 + [0 : N - 1] - N = [1 : N] - N = [1 - N : 0]$, which must be zero or negative, allowing STLlint to conclude that $1 + (x \in [0 : N - 1]) \leq N$.

4.4. Induction variable recognition

The symbolic representation of integer values within STLlint is crucial to the application of induction variable recognition. As demonstrated in Fig. 6, we require recognition of induction variables that are accessed via multi-level pointers and may be assigned multiple times within a single loop iteration. In fact our induction variables are not generally variables at all, but are fields of objects that are modified through pointers in subroutine calls. We therefore refer to *induction locations*, i.e., abstract memory locations that represent memory locations that hold integral values. Each field of a particular object is associated with a unique abstract memory location, so that by performing induction variable recognition on the abstract locations we can effectively deduce the inductive behavior of these locations even when they are accessed via multi-level pointer expressions.

4.4.1. Symbolic differencing

The use of induction locations in lieu of induction variables drastically reduces the potential set of existing induction variable recognition algorithms that we may employ. Induction variable recognition algorithms that detect patterns in the definitions of integer variables [35, 36], most notably those that are based on Static Single Assignment (SSA) form [37, 38], are unusable in this context

```
template<typename InputIterator, typename Predicate>
  InputIterator
  find_if(InputIterator first, InputIterator last, Predicate pred)
  {
    while (first != last && !pred(*first)) { ++first; }
    return first;
  }
```

Figure 7. Implementation of the generic `find_if` algorithm, which searches for the first element in a sequence that satisfies the given predicate.

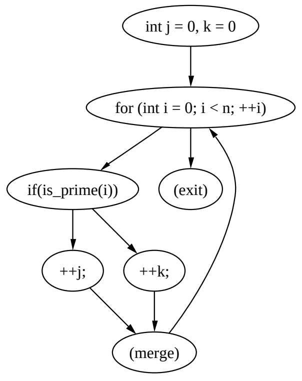
because they operate on integer variables only and cannot cope with integer locations accessed via pointers. Symbolic differencing [39], on the other hand, can recognize the inductive behavior of integer locations, and is therefore the method of choice in STLlint. Symbolic differencing applies Newton's forward formula for interpolation to recognize generalized induction expressions of the form $\chi(n) = \varphi(n) + ar^n$ for a polynomial φ (with loop-invariant coefficients) and loop-invariant a and r, where the maximal degree of the polynomial, n, is a parameter to the symbolic differencing analysis. Symbolic differencing requires only that the analysis symbolically execute the loop body in its most general form (i.e., by replacing the value at each integer location with a fresh symbolic constant), iterating $n + 2$ times and recording the values of each integer location after each iteration. Thus given a suitable points-to analysis that can associate arbitrary pointer expressions with integer locations, STLlint can derive induction expressions for integer locations regardless of how—or where—integer locations are accessed. In an iterator loop such as that of Fig 7, the location `first.position` will originally be assigned a fresh symbolic variable p, and will assume the values $p{+}1, p{+}2, ..., p{+}n{+}2$ at the end of the $n{+}2$ loops. Applying symbolic differencing to this sequence $(p, p{+}1, p{+}2, ..., p{+}n{+}2)$, we derive the induction expression $p_0 + i$ for the integer location `first.position`, where p_0 is the initial value of `first.position` and i is the iteration number.

4.4.2. *Trip count calculation*

The generality of symbolic differencing also enables STLlint to accurately determine trip counts, i.e., the number of times the loop body will execute, even when we cannot easily relate the loop termination condition to the values of induction variables before or after a particular loop iteration. In Fig. 7, the two parts of the loop termination condition, `first != last` and `!pred(*first)`, both involve subroutine calls that may have side effects and are separated by a conditional branch due to the short-circuit evaluation of the `&&` operator. Fig. 9 makes these branches and function calls more explicit by "lowering" the `find_if` implementation, illustrating that we cannot simply use the values of the integer locations compared via `first != last` to determine the trip count, leaving the static analyzer with two options: either prove that the integer locations involved in the expression `first != last` are unchanged by the subsequent call to `pred`, or attempt to compute the inductive behavior of `first != last` without relying on the induction expressions computed for the requisite integer locations. STLlint utilizes the latter option, by computing the set of integer expressions involved in the

```
int j = 0, k = 0;
for (int i = 0; i < n; ++i)
    if (is_prime(i))
        ++j;
    else
        ++k;
}
```

(a) Monotonic induction variables



(b) Control flow graph

Figure 8. A simple loop containing a basic induction variable i and two monotonic induction variables, j and k.

loop termination condition. The values of these expressions are recorded at each loop iteration (as we have done for integer locations), and symbolic differencing computes induction expressions for these expressions that will be used in trip count calculations. The technique used to isolate the set of integer expressions when they occur within subroutines will be described in Sec. 4.5.1. As a peculiar side benefit, calculating induction expressions for expressions in the loop termination condition allows us to determine trip counts even in certain cases where the integer locations involved in the termination condition are not themselves inductive [39].

4.4.3. Monotonic induction variables

STLlint implements monotonic induction variables within the context of symbolic differencing by careful introduction of additional symbolic variables during the control-flow merge operation. Control-flow merges occur in flow-sensitive but path-insensitive analysis algorithms, where on two separate paths the same variable or location possesses two different values; when these two paths rejoin, perhaps at the end of an `if–then–else` construct, the analysis must produce a new value for that variable or location that approximates the values on both paths. With traditional value range propagation [33], this merge operation computes the resulting value range $[a : b]$ where a is the minimum of all incoming values and b is the maximum of all incoming values. For instance, if a variable m is assigned the value x in one path, and the value $x + 1$ in another path, then the resulting integer value of that variable after control-flow merge of the two paths will produce the value range $[\min(x, x + 1) : \max(x, x + 1)] = [x : x + 1]$.

The introduction of value ranges at merge points hampers the application of symbolic differencing for that location, because symbolic differencing is formulated based on symbols alone, not value ranges. The loop in Fig. 8 contains a monotonic variable, j, whose maximum value is bounded by n. Using the traditional value range merge operation, j would take on the values $j_x, [j_x : j_x + 1], [j_x : j_x + 2], [j_x : j_x + 3], [j_x : j_x + 4]$ through successive iterations, where j_x is an arbitrary symbolic variable used for induction variable recognition. Symbolic differencing cannot derive the inductive behavior of j from this sequence of values.

We note that in Fig. 8, the value of j is increased by either 1 (when i is prime) or 0, when i is not prime. Thus instead of introducing a new value range within the control-flow merge operation, we produce an equivalent symbolic value by capturing the variation in a fresh symbolic constant $\alpha \in [0 : 1]$, and declare the result of the value range merge operation to be $j_x + \alpha$. In the example of Fig. 8, α describes the possible changes to the variable j based on the value of `is_prime(i)` in a given iteration. The use of α in successive iterations therefore refers to the same *event*, but not to a particular constant value (only the range is constant). By reusing α when merging values of j at the control-flow join within this loop, we determine that j will take on the values $j_x, j_x + \alpha, j_x + 2\alpha, j_x + 3\alpha, j_x + 4\alpha$. Symbolic differencing can then be applied to this sequence to produce an induction expression $j_0 + i\alpha$ for the variable j, where i is the iteration number and $\alpha \in [0 : 1]$ is loop-invariant. Thus we can determine that on the k^{th} iteration of the loop, the value of j will be its initial value j_0 plus αk, i.e., some value in $[0 : k]$. This result is more precise than could be determined via abstract interpretation with widening and narrowing [29], which would determine only that the value of $j \geq j_0$ because widening does not apply when j is not used in the loop termination condition.

Our formulation of monotonic variables is sufficient for some, but not all instances of monotonic variables we have encountered in the use of STLlint. In particular, the iterator-erasure example in Fig. 6 requires that the two monotonic variables—`iter.position` and `students.size`—be related so that the invariant `iter.position` $\leq$ `students.size` can be verified. STLlint further enhances the control-flow merge operation to retain such a relation by exploiting knowledge of important relationships amongst fields in different objects; this algorithm is the subject of another paper [40].

4.5. Interprocedural loop analysis

STLlint addresses the need for interprocedural loop analysis in several ways. The most important aspect of this support, that of representing the values at integer locations via symbolic expressions (Sec. 4.3) and applying symbolic differencing to these locations (Sec. 4.4.1), has already been discussed. Our approach to loop analysis with symbolic execution naturally supports interprocedural induction location recognition. However, induction location recognition itself does not suffice for loop analysis: we require precise interprocedural trip count calculation, discussed in Sec. 4.5.1, and also benefit from techniques that simplify the loop termination condition, as discussed in Sec. 4.5.2.

4.5.1. Interprocedural trip count calculation

The trip count calculation described in Sec. 4.4.2 requires that one calculate the set of integer expressions that are involved in the termination condition. We first describe the representation of this set, and then present the construction and evaluation of a loop termination condition using the members of this set.

Integer expressions involved in the loop termination condition must be uniquely located within the abstract syntax tree and associated with a particular chain of subroutine invocations. Thus if the

```
iterator find_if(iterator first, iterator last, Predicate pred)
{
  bool b1 = not_equal(&first, &last);
  bool b2;
  if (b1) {
    T* ref = deref_iterator(&first);
    bool b3 = pred(ref);
    b2 = !b3;
  }
  if (b1 && b2) {
    do {
      iterator_increment(&first);
      bool b4 = not_equal(&first, &last);
      bool b5;
      if (b4) {
        T* ref = deref_iterator(&first);
        bool b6 = pred(ref);
        b5 = !b6;
      }
    } while (b4 && b5);
  }
  return first;
}

bool not_equal(iterator* x, iterator* y)
{
  semple_assert(x->sequence && x->position <= x->sequence->position);
  semple_assert(y->sequence && y->position <= y->sequence->position);
  semple_assert(x->sequence == y->sequence);
  return x->position != y->position;
}
```

Figure 9. Partial expansion of the `find_if` function from Fig. 7, illustrating the static analyzer's view of the "simple" iterator loop.

expression $x + 1$ in a function f is evaluated twice within the loop body because f is invoked from two different call sites in the loop, the two evaluations are considered distinct from the point of view of the set of integer expressions. STLlint represents these integer expressions as $(context, expression)$ pairs, where the $context$ refers to the node within the invocation graph [41] at which the expression is evaluated and $expression$ refers to the node in the abstract syntax tree that represents the expression. The use of an invocation graph allows STLlint to efficiently represent and compare chains of subroutine invocations.

Fig. 9 contains a partially-expanded implementation of the `find_if` algorithm originally presented in Fig. 7, such that function calls and control flow have been made more explicit. The loop termination condition present in the `do`–`while` loop references two boolean variables, from which we cannot directly derive a trip count. Instead, we must trace these boolean variables back to their definitions to uncover the integer relations that govern loop termination: `b4` can be traced back to the result of the call to `not_equal`, which in turn is the result of the integer relation `x->position != y->position`, whereas `b5` can be traced to the logical negation of `b6`, that itself is the result of a call to the unknown predicate `pred`. Denoting chains of subroutine calls via $s_1 : s_2 : \cdots : s_n$, our true loop termination condition is

$$(\texttt{find\_if} : \texttt{not\_equal}, \texttt{x->position}) \neq (\texttt{find\_if} : \texttt{not\_equal}, \texttt{y->position}) \wedge unknown.$$

Since we do not have any information at this time to approximate the result of `pred(ref)`, we instead apply the placeholder $unknown$ that represents an unknown termination condition. STLlint constructs the loop termination condition by computing an SSA numbering [37, 38] for each subroutine, so that boolean variables occurring in the termination condition may be (recursively) replaced with the appropriate definition. Variables whose definition is the result of a function call will be replaced with the return expression within the call context, after the callee has been normalized to contain only a single return statement.

The values of each integer expression $(context, expression)$ evaluated in the true loop termination condition are recorded for $n + 2$ iterations of the loop body, and symbolic differencing is applied to compute induction expressions for each. The trip count is computed from the loop termination condition using the following rules:

- Comparisons between a linear induction expression and a loop invariant results in an exact trip count.
- Comparisons between two linear induction expressions results in a trip count between 0 and the distance between the initial values of the two induction expressions (if they converge), or between 0 and ∞ (if they diverge).
- $unknown$ expressions result in a trip count between 0 and ∞.
- Logical conjunctions $x \wedge y$ result in the minimum trip count computed for the subexpressions x and y.
- Logical disjunctions $x \vee y$ result in the maximum trip count computed for the subexpressions x and y.

The trip count for our example in Fig. 9, given the initial value 0 for $(\texttt{find\_if} : \texttt{not\_equal}, \texttt{x->position})$ and N for the initial value of $(\texttt{find\_if} : \texttt{not\_equal}, \texttt{y->position})$, will be computed as $\min(N, [0 : \infty]) = [0 : N]$.

4.5.2. Goal-directed inlining

STLlint performs interprocedural loop trip count determination to accurately analyze the behavior of program loops with higher-level iteration constructs. Another problem that plagues static analyzers when faced with such loops is that flow-sensitive information generated from conditional branches is not easily propagated. For instance, the boolean variable `b4` contains the result of the expression `x->position != y->position`, where `x` is the address of `first` and `y` is the address of `last`. Within the `then` branch of the conditional involving `b4`, it is guaranteed that `first.position != last.position`, but it is nontrivial to propagate the information from the source of `b4`'s value—inside the function `not_equal`—to the use of `b4` in a conditional branch. It is essential that this

```
do {
  iterator_increment(&first);
  semple_assert(first.sequence
              && first.position <= first.sequence->position);
  semple_assert(last.sequence
              && last.position <= last.sequence->position);
  semple_assert(first.sequence == last.sequence);
  bool b4 = first.position != last.position;
  bool b5;
  if (first.position != last.position) {
    T* ref = deref_iterator(&first);
    bool b6 = pred(ref);
    b5 = !b6;
  }
} while (b4 && b5);
```

Figure 10. “Optimized” form of the `do`–`while` loop from Fig. 9, after boolean variable definitions have been “pulled” into the `if` condition.

information be propagated: the body of the `deref_iterator` operation (not shown) contains an assertion that requires `first.position != last.position`.

To aid in the propagation of information present in conditional branches, STLlint performs goal-directed inlining to “pull” the underlying definitions of boolean variables used in conditional branches into the branch conditions. The process begins by computing SSA numbers for all boolean variables in the function; then, the calls defining any boolean variable referenced within an `if` or a `do`–`while` condition are inlined. Finally, the resulting code is optimized via copy propagation and the elimination of unused variables. Fig. 10 illustrates the result of pulling the conditional expression for `b4` into the `find_if` loop; we see that this result enables STLlint to propagate the assumption `first.position != last.position` into its `then` branch, guaranteeing that the assertion within the iterator dereference operation will not produce a false positive.

4.6. Putting it all together

Analysis of the example in Fig. 6 relies primarily on precise loop analysis. In the initial phase of translation, the calls to library routines and the definitions of library data structures are replaced by their respective executable specifications. Prior to the loop, symbolic execution determines that `iter` references the `students` vector and has a position of zero (the beginning of the vector), the `fail` vector has size zero, and we assume for this discussion that the `students` vector is nonempty.

When the analysis encounters the loop, it forks a separate program state for loop analysis. We employ fixed-point iteration (as in abstract interpretation) for noninductive variable types (e.g., pointers). In the resulting state, the values for all integer locations are replaced with fresh symbolic constants, and the loop is executed several times, recording the values of (1) each integer location (or variable), and (2) each subexpression involved in the termination condition, at each iteration. Symbolic differencing then determines that the position of `iter` is monotonically increasing (by 0 or 1 each iteration), the size of `students` (and therefore the position of `students.end()`) is monotonically decreasing and changing when the position of `iter` does not change [40], and finally that the size of `fail` is monotonically increasing. Symbolic differencing is then applied to the values of each termination condition subexpression to compute a loop trip count.

Analysis of the loop completes by replacing the symbolic constants used in symbolic differencing by the initial values of the pre-loop state. Here we determine that the loop executes n times, where n is between 1 and `students.size()`. The loop is then symbolically executed in its own forked program state using the induction expressions (e.g., the position of `iter` is $\alpha \cdot i$, where i is the loop iteration number and $\alpha \in [0 : 1]$) and allowing i to vary between 0 and $n - 1$. Thus, we are checking all iterations of the loop in one symbolic execution of the loop body. STLlint is therefore able to verify the correctness of all assertions in all iterations of the loop.

The side effects of the loop are determined by again replacing the symbolic constants introduced for symbolic differencing by the initial conditions, and replacing the values of integer locations with induction expressions. The loop body is then executed once with $i = n-1$, simulating the final iteration of the loop.

5. Organization of an extensible static checker

Higher-level checking involves not only checking of higher-level constructs (e.g., containers and iterators) but also higher-level semantic properties, such as whether the values in a sequence have been sorted or whether the values of a certain subsequence cannot be relied upon to be accurate. The C++ standard [6] specifies the requirements of many STL algorithms in terms of higher-level semantic properties, in the manner shown in Fig. 1. We do not attempt to construct proofs that algorithms introduce certain properties, instead relying on hand-written algorithm specifications that assert the appropriate semantic properties. STLlint specifications provide a method of "tagging" objects with other objects of varying types and accessing those "tag" objects later in the program. Specifications may create, query, or destroy these tags (that may themselves carry additional information) at any point, allowing for instance a sorting algorithm to introduce the "sorted" tag (coupled with the ordering relation), that will be verified any time the sequence is required to be sorted (e.g., when one calls a binary search function such as `lower_bound`), and that will be destroyed by any attempt to modify the sequence that doesn't explicitly preserve sortedness. Fig. 11 illustrates an example where the author has omitted the proper sorting invariant, for which STLlint produces a reasonable diagnostic:

```
"sort_insert_insert.cpp", line 21, warning: sequence may have been
      sorted with a different predicate than the one given

    i = lower_bound(v.begin(), v.end(), 17);

in call to function lower_bound at "sort_insert_insert.cpp", line 21
```

Even by limiting ourselves to semantic properties useful within the C++ STL, there is a large number of combinations that must be handled. The STL contains more than 70 algorithms and 8 different container types, with many interesting interactions among them. We must therefore consider

```
vector<int> v;
// fill v
sort(v.begin(), v.end(), greater<int>());
vector<int>::iterator i =
  lower_bound(v.begin(), v.end(), 42, greater<int>());
v.insert(i, 42);
i = lower_bound(v.begin(), v.end(), 17);
v.insert(i, 17);
```

Figure 11. A small code snippet that improperly attempts to insert two values into a sorted sequence. The second `lower_bound` invocation does not use the same ordering relation as was previously used to sort the sequence.

the incremental cost of introducing checking for a single new semantic property. For instance, to add proper checking for sortedness in such a system, we would need to:

- Annotate all sorting functions to state that they make the sequence "sorted", and
- annotate all functions that require a sorted sequence to perform checking of the "sorted" attribute, and
- annotate all functions that may modify or reorder the sequence so that they modify or remove the "sorted" attribute.

In essence, the need to maintain tags means that the addition of a single new function requires one to reexamine the entire system of specifications, drastically hampering extensibility. To address this problem, we define a set of *algorithm concepts* [42] that describe the behavior of algorithms and allow STLlint to reduce the cost of introducing new semantic checks. We describe algorithm concepts in Sec. 5.1 and detail the implementation of algorithm concepts in STLlint in Sec. 5.2.

5.1. Algorithm concepts

Within STLlint, we are primarily concerned with the behavior of algorithms with respect to the (iterator) sequences on which they operate. Algorithm concepts categorize the high-level semantics of algorithms, for instance grouping all sorting algorithms together under the SORTING concept or grouping all binary search algorithms together under the SORTEDSEARCHING concept. Algorithm concepts range from very unspecific, very general concepts such as READs a sequence or WRITEs a sequence, to very specific concepts describing the behavior of particular algorithms, such as a heap sort.

Semantic properties are associated with algorithm concepts by providing specifications of the behavior of algorithm concepts with respect to that semantic property. For instance, to perform checking related to sorting we specify that the SORTEDSEARCHING concept check that the incoming sequence is sorted; the SORTING concept assert that the sequence as sorted; and the WRITE concept assert that the sequence is no longer guaranteed to be sorted. We refer to a semantic property being checked as an *axis*, and here we have informally specified the semantics of different algorithm concepts along the "sortedness" axis. To differentiate the specification of algorithm concept behavior along an axis from the specification of data structures and algorithms, we apply the term *customization* to the

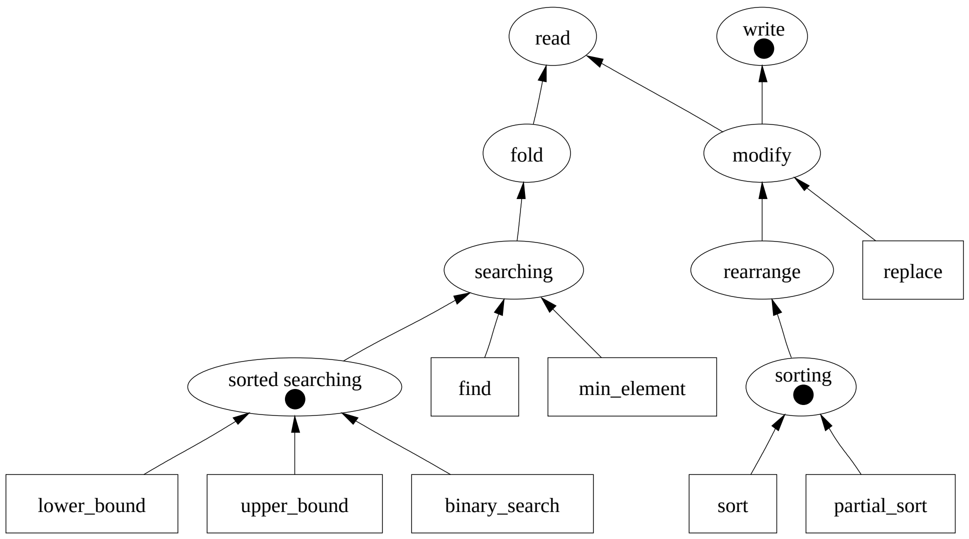


Figure 12. Partial algorithm concept lattice as used in STLlint

former. STLlint supports the introduction, removal, and verification of semantic property tags (such as "sorted") based not on algorithms but algorithm concepts. That is, all algorithms that model the SORTING concept will introduce the "sorted" tag to assert sortedness, SORTEDSEARCHING algorithms will verify the existence of the "sorted" tag on the input sequence, and any other algorithm that WRITEs an iterator sequence will remove the "sorted" tag. The ellipses containing dots in Fig. 12 represent the customization points required to implementing checking of "sorted" properties. The details of the decision process that supports these semantics require additional relationships among algorithm concepts.

Algorithm concepts are arranged within a concept lattice [43], where the least specific concepts are placed at the top of the lattice with the most specific concepts at the bottom. The (implicitly) directed edges represent the *refinement* relation, where the statement "A *refines* B" indicates that the concept A inherits all properties of concept B but also introduces its own properties. Fig. 12 contains a partial algorithm concept lattice. For instance, note that MODIFY—meaning that the algorithm modifies a particular sequence— refines both READ and WRITE. Furthermore, the REARRANGE concept refines the MODIFY concept because rearranging the values in a sequence modifies the sequence, but more specifically it performs a reordering of the elements instead of arbitrarily modifying them.

To determine the effects of an algorithm, STLlint finds the most specific algorithm concept(s) modeled by the algorithm and customized along some axis. The search for the most specific algorithm concept(s) modeled by a particular algorithm is a depth-first search starting at the concept for that particular algorithm (represented by a rectangle) in the lattice and proceeding up through other algorithm concepts (represented by ellipses) in the lattice until a concept customized by that axis is found along that branch. For instance, the search for a customization point for the `replace` algorithm along the sortedness axis would find the WRITE concept (i.e., the first dot found when traversing up the lattice). Thus the author of `replace`, by merely stating that the algorithm models the MODIFY concept, has in effect stated that replacing elements in a sorted sequence may result in an unsorted sequence. If a concept refines two or more concepts, it is possible that several customization points may apply for a particular axis and algorithm concept. Additionally, since the customization of concepts occurs on several different axes (for multiple, distinct semantic property checks), potentially many customization points may apply for any given algorithm.

Algorithm concepts, and particularly the concept refinement relationship, provide a way to decouple algorithms from semantic checks on the algorithms, drastically reducing the effort required to introduce extensions to STLlint. Introducing a new algorithm requires only that one state the concepts that the algorithm models, after which the algorithm seamlessly performs checks and updates of the various semantic tags. Introducing a new semantic check (axis) requires one to isolate the concepts of interest within this axis and specify the assertions and effects that algorithms modeling these concepts have on the semantic tags involved. While the effort expended to isolate and specify the behavior of algorithm concepts is nontrivial, the process involves much less redundancy than the alternative, and provides the additional benefit of being modular: STLlint allows one to toggle the checking of various semantic properties from the command-line interface.

5.2. Algorithm events

STLlint provides an extensible implementation of algorithm concepts and axes. Algorithm concepts themselves are represented by otherwise empty C++ class types, with concept refinement represented by inheritance. Similarly, axes are represented by C++ class types and a set of C++ functions that operate on certain algorithm concepts. The C++ manifestations are not a part of the static analyzer itself, but are part of STLlint's modified C++ standard library implementation, compiled along with—and therefore customizable by—the user's code. An "active" C++ library [44] may introduce its own axes, algorithm concepts, algorithms, and data structures that STLlint can then analyze.

Each algorithm must explicitly state the algorithm concept it models, providing information about its arguments and return value. This information is provided to STLlint's *event* mechanism that enables the customization of behavior along each axis for algorithm concepts. Customization is permitted when the algorithm has been invoked (via the `on_entry` event) and when the algorithm has completed its computation (via the `on_exit` event), allowing arbitrary executable specifications to, e.g., check preconditions and assert postconditions. Fig. 13 illustrates the STLlint directive that associates the `lower_bound` algorithm with its algorithm concept; again, the event mechanism implementing algorithm concepts in STLlint is standard C++ code.

Customizations are implemented as C++ function templates (called event *handlers*) whose function parameters correspond to the algorithm concept they customize (e.g., SORTEDSEARCHING), the axis along which they perform the customization (e.g., "sortedness"), and the function parameters provided by the algorithm to the event. Fig. 14 illustrates the `on_entry` handler responsible for uncovering the inconsistent predicate in Fig. 11. Note that this single entry point applies to the `lower_bound`, `upper_bound`, `equal_range`, and `binary_search` algorithms from the C++ standard library, without requiring one to annotate the algorithms separately. Customizations of `on_exit` events are

```
template<typename ForwardIterator, class T, typename Compare>
  ForwardIterator
  lower_bound(ForwardIterator first, ForwardIterator last,
              const T& value, Compare compare)
  {
    _STLlint::event<
      __events::lower_bound,
      ForwardIterator(ForwardIterator, ForwardIterator, const T&, Compare)
      > ep(first, last, value, compare);

    ForwardIterator result;
    // ...
    return ep(result);
  }
```

Figure 13. Skeletal implementation of the `lower_bound` algorithm including STLlint's annotation associating the implementation with the LOWER_BOUND concept.

similar, with one exception: the return value is available as a parameter to the `on_exit` function directly following the axis, to allow the handler to assert postconditions on the result.

STLlint's events and event handlers provide a means to decouple algorithms from their pre- and post-conditions effectively via algorithm concepts. However, this decoupling introduces additional levels of abstraction that place semantic checks far from the algorithms whose preconditions they verify, complicating the process of generating diagnostics relating directly to the algorithm invocation. For instance, error messages that are written within event handlers refer to the parameters of the algorithm concept they customize and not the arguments actually passed to the algorithm. For this reason, the event dispatching mechanism constructs a mapping from the parameters of an algorithm concept to the arguments of the concepts it refines; when a diagnostic is required, STLlint follows the argument mappings in reverse to associate the parameters referenced within an event handler to the actual arguments the user passed to a particular algorithm, resulting in diagnostics such as the one presented in Sec. 5.

6. Related work

Engler presents the MAGIK [8] open compilation system that allows the programmer to write dynamically-loaded modules that operate directly on the C compiler's internal representation (IR). These modules may inspect, transform, and even optimize the program during compilation, permitting additional checking and optimization for software libraries. MAGIK is more extensive than STLlint because it permits transformation of the program, but the direct interface to the compiler's IR hampers portability and does not scale well to more complex languages and libraries due to the greater variance and complexity of the IR. Similarly, OpenC++ [45] implements a "meta object protocol" for C++, permitting objects that exist only at compile time to direct compilation, and has been applied to the

```
template<typename ForwardIterator, typename Compare>
  void
  on_entry(sorted_searching, sortedness, ForwardIterator first,
           ForwardIterator last, Compare comp)
{
  semple_assert(sortedness::requires(first, last),
                "sequence not sorted...");
  semple_assert(sortedness::requires(first, last, comp),
                "sequence sorted with different predicate...");
}
```

Figure 14. Implementation of the `on_entry` event along the “sortedness” axis for algorithms modeling the SORTEDSEARCHING concept. This function template verifies that all sorted searching algorithms receive sequences sorted with the same ordering operation `comp`.

checking and implementation of design patterns in C++ [46]. If one gives up static safety guarantees, the specification of class invariants and pre- and post-conditions that Eiffel introduced ([47], also see [48]), provides alternative means for (dynamically) checking safety, yet lacks the precision of control-flow analysis that iterator-based programs often require.

The problem of iterator invalidation is as old as the concept of an iterator itself, but is defined in different ways for different languages. CLU [49], the language that introduced the term “iterator”, was limited to a single iterator per loop and prohibits the modification of a collection while an iterator is active. While these design decisions greatly improved the potential for static analysis on iterator traversals, these limitations are overly restrictive (see, e.g., [50]). Similarly in the Java Collections Framework (JCF) [26], where iterators are invalidated by any operation on the container that does not occur through the iterator, thereby allowing the problem of invalidation to be reduced to a versioning problem. The semantics of iterators in the C++ STL, in contrast, are more deeply tied to the underlying data structures, and are only invalidated when the memory they reference may no longer be available: for instance, a `vector` iterator will be invalidated by an erasure from the container only if the iterator references the erased element or any element after it whereas erasing from a `list` invalidates only iterators referencing the element being erased. Depending on the iterator semantics, the complexity of the analysis varies. For the JCL, Ramalingam et al. describe a method of deriving program analyses [7] suitable for verifying component-client conformance, which they apply to (what they call) the Concurrent Modification Problem (CMP), i.e., the use of an iterator that references a container that has been modified. They also show the need for deriving specialized analyses for problems such as CMP, for which general static analyses are impractical for large programs. On the other hand, the more-complex invalidation semantics of the C++ STL, and the ability of some STL iterators to move in either direction (Bidirectional iterators) or an arbitrary number of steps (Random access iterators) mandate that STLlint utilize a more expressive form for iterators than is needed for iterators in the Java Collections Framework.

7. Evaluation

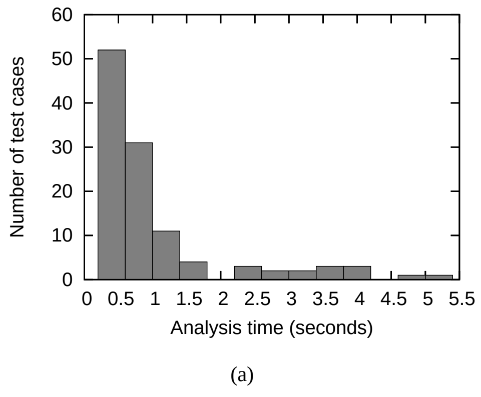
We constructed a new test suite to evaluate STLlint, because no existing test suite exercised the unique properties of high-level checking for the STL. Our test suite consists of roughly 110 tests, the vast majority of which have been collected from the following sources:

- *Accelerated C++* [27]: This book teaches introductory C++ using STL. The example code presented throughout the book therefore exhibits precisely the style of programming targeted by STLlint, providing many negative tests (i.e., those without errors). Furthermore, the book provides several examples of nearly-correct code that fails due to iterator invalidation; these examples are included as positive tests that also provide motivation for STLlint.
- *Effective STL* [15]: This book describes various pitfalls with the use of STL. We have adapted some of the examples that correspond to interesting properties checked by STLlint into both positive and negative test cases.
- *GNU libstdc++ test suite* [51]: This test suite is meant to exercise the GNU implementation the C++ standard library. As of version 3.4 of the GNU compiler, the test suite includes (negative) tests for correctness of the implementation and (positive) tests that verify correct behavior of the run-time checked version of the library. The tests relevant to the STL have been incorporated into the STLlint test suite.

The STLlint static checker performs favorably, achieving a 0.59% false positive rate on this test suite. We define a false positive as any warning produced by STLlint where manual inspection of the source code has concluded that the source code is, in fact, correct. We verified our findings with tests on the complete source code from another STL reference [9], achieving a 2.19% false positive rate. Further inspection indicates that 49% of these false positives were due to expectations on the input not calculable by STLlint. For instance, a sample program might call the STL `find` algorithm on a known input and would then dereference the resulting iterator without checking that it is dereferenceable. STLlint produces a false positive because it does not model the hard-coded input in its analysis. Another 25% of the false positives are due to the use of "lower-level" C++ constructs, such as C-style arrays and string literals, which are not accurately modeled by STLlint. For instance, the length of a string literal is lost when that string is passed to a function via a character pointer. The project web page [52] gives a complete analysis of these false positives.

To assess the efficiency and scalability of STLlint, we focus on the time required to analyze the programs within the STLlint test suite. We then compare that time against the time required to parse and type check the program, where a smaller ratio of analysis time to parsing/type checking time indicates a more scalable analysis relative to program size. All data was gathered on a system using an AMD AthlonXP 2500+ processor with 512 megabytes of RAM running under version 2.4.22 of the Linux kernel. STLlint itself employs version 3.0.1 of the Edison C++ front end [21] and was compiled with version 3.3.2 of the GNU C++ compiler [51] with a high optimization level (`-O3`).

The histogram in Fig. 15(a) illustrates the analysis time required by examples in our test suite, most of of which are analyzed in under a second. To scale these results by program size, Fig. 15(b) provides a histogram of the ratios of analysis time to the time spent in the front end. While the majority of the test cases require less time to analyze than to parse and type check, there are a few notable exceptions where the analysis runs ten times longer than the front end. We have observed that the analysis time



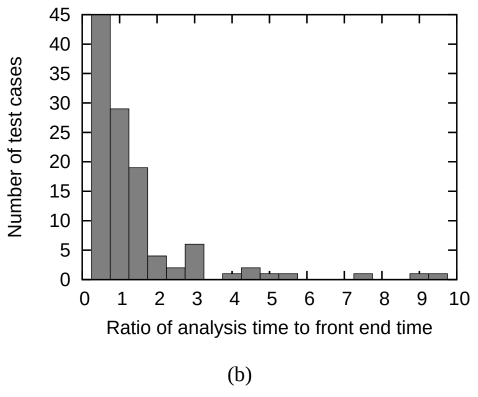


Figure 15. Histograms illustrating the analysis time (in seconds) required by STLlint and the ratio of analysis time to front end time for our STLlint test suite

is proportionate to the number of integer comparisons performed by STLlint, which is not directly correlated to program size: specifications of STL `set` and `map` insertion operations, for instance, introduce many more integer comparisons than the corresponding operations in other data structures. Not coincidentally, `set` and `map` insertion operations also degrade precision in certain instances [32, §8.4.1], resulting in several false positives within the STLlint test suite. Future work will focus on these particular weaknesses and may result in improvements for both precision and performance.

8. Conclusion

Programmers benefit from the static checking tools that can diagnose program errors early in the development cycle. For static checkers to be useful to the programmer, they must operate at or near the same level of abstraction as the source code itself. We have presented the challenges of constructing a static checker, STLlint, that performs checking at the abstraction level of the C++ standard library and is suitable for the vast majority of examples in two introductory C++ textbooks [27, 9]. Our experience has been that checking higher-level abstractions is more complex than checking lower-level languages, as many important analyses—such as loop analysis—become drastically more complicated when presented with additional layers of abstraction. We also found that we can counteract these complicating factors by replacing abstractions with simpler models that reduce overall complexity. We therefore assert that higher-level, library-centric analysis is markedly different from a lower-level, language-based analysis, because the former requires one to embrace the abstractions crucial to the programmer's understanding of the problem domain.

8.1. Availability

The Semple static analysis engine consists of approximately 15k lines of C++ source code, augmented by the GiNaC symbolic computation framework [53], implementing all of the algorithms described here and in our companion paper [40]. It provides support for the analysis techniques we have needed in STLlint, through a generalized symbolic execution framework. The static analysis engine, including tools to manipulate programs written in the Semple language, is available for download under an open-source license [54].

The high-level specifications for the C++ Standard Template Library, which comprise roughly 6k lines of C++ code, describe the semantic behavior of all containers, iterators, and algorithms within the STL, as specified in the C++ Standard [6]. These specifications mimic precisely the interfaces of the STL components they replace, and employ a small set of primitives provided by the C++ parser to support the introduction of Semple constructs not otherwise expressible in C++. As with the Semple analysis engine, these specifications are freely available [52].

The link between the Semple static analysis engine and the high-level STL specifications is the transformation from the internal representation of the Edison C++ front end [21] to the Semple intermediate representation. While the front end itself is proprietary, the corresponding module, of approximately 3.5k lines of C++ code, can be replaced with a similar transformation for a different compiler's front end with only a moderate amount of effort. For users interested in using STLlint as-is, we have provided an online system where users can submit C++ source code and receive STLlint's diagnostics [52].

ACKNOWLEDGEMENTS

We thank the Edison compiler group for making their front-end available, thus saving us the trouble of parsing C++ ourselves. Two anonymous reviewers made helpful suggestions regarding the empirical part and the presentation of the paper.

REFERENCES

1. S. C. Johnson. Lint, a C program checker. Technical Report 65, AT&T Bell Laboratories, 1978.
2. David Evans. Annotation-assisted lightweight static checking. In *First International Workshop on Automated Program Analysis, Testing and Verification*, February 2000.
3. David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the SIGSOFT Symposium on the Foundations of Software Engineering*, pages 87–96. ACM, 1994.
4. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–241, 2002.
5. Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1998.
6. ANSI-ISO-IEC. *C++ Standard, ISO/IEC 14882:1998*, ANSI standards for information technology edition, 1998.
7. G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 83–94, June 2002.
8. Dawson R. Engler. Interface compilation: Steps toward compiling program interfaces as languages. *Software Engineering*, 25(3):387–400, 1999.
9. David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library, Second Edition*. Addison-Wesley, 2001.

10. Alexander A. Stepanov and Meng Lee. The Standard Template Library. Technical Report HPL-95-11, Hewlett Packard, November 1995.
11. David Musser and Alexander Stepanov. Algorithm-oriented generic libraries. *Software–Practice and Experience*, 27(7):623–642, July 1994.
12. Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1999.
13. David Musser and Alexander Stepanov. *The ADA Generic Library: Linear List Processing Packages*. Springer-Verlag, 1989.
14. Douglas Gregor and Sibylle Schupp. Making the usage of STL safe. In Jeremy Gibbons and Johan Jeuring, editors, *Generic Programming, IFIP TC2/WG2.1 Working Conference on Generic Programming*, volume 243 of *IFIP Conference Proceedings*, pages 127–140. Kluwer, July 2003.
15. Scott Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley, 2001.
16. Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–15, January 1996.
17. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updatin g. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 16–31, January 1996.
18. Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *Proceedings of the First Workshop on C++ Template Programming*, October 2000.
19. Jeremiah Willcock, Jeremy Siek, and Andrew Lumsdaine. Caramel: A concept representation system for generic programming. In *Proceedings of the Second Workshop on C++ Template Programming*, October 2001.
20. Leor Zolman. An STL error message decryptor for Visual C++. *C/C++ User's Journal*, 19(7):24–30, July 2001.
21. Edison Design Group C++ front end. `http://www.edg.com/`.
22. Standard Template Library programmer's guide. `http://www.sgi.com/tech/stl/`, 2003.
23. Sergio Antoy and Dick Hamlet. Automatically checking an implementation against its formal specification. *Transactions on Software Engineering*, 26(1):55–69, January 2000.
24. Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 121–142. Kluwer Academic Publishers, 1996.
25. Changqing Wang and David R. Musser. Dynamic verification of C++ generic algorithms. *Software Engineering*, 23(5):314–323, 1997.
26. Patrick Chan, Douglas Kramer, and Rosanna Lee. *The Java Class Libraries: Supplement for the Java 2 Platform*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1999.
27. Andrew Koenig and Barbara E. Moo. *Accelerated C++*. Addison-Wesley, 2000.
28. François Bourdoncle. Efficient chaotic iteration strategies with widenings. *Lecture Notes in Computer Science*, 735:128–142, 1993.
29. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.
30. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.
31. Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice Hall, Englewood Cliffs, New Jersey, 1981.
32. Douglas Gregor. *High-Level Static Analysis for Generic Libraries*. PhD thesis, Rensselaer Polytechnic Institute, April 2004.
33. William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.
34. William Blume and Rudolf Eigenmann. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 357–363, April 1995.
35. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
36. Michael Wolfe. Beyond induction variables. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 162–174, 1992.
37. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–11, 1988.
38. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 12–27, 1988.

39. Mohammad R. Haghighat and Constantine D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996.
40. Douglas Gregor and Sibylle Schupp. Retaining path-sensitive relations across control-flow merges. Technical Report 03-15, Rensselaer Polytechnic Institute, November 2003.
41. Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 242–256, 1994.
42. Sibylle Schupp, Douglas Gregor, Brian Osman, David R. Musser, Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. Concept-based component libraries and optimizing compilers. Technical report, RPI Computer Science Department Technical Report 02-02, 2002.
43. Rudolf Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In Ivan Rival, editor, *Ordered Sets*, pages 445–470. NATO Advanced Study Institute, September 1981.
44. Todd L. Veldhuizen and Dennis 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)*, 1998.
45. Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 285–299, October 1995.
46. Michiaki Tatsubori and Shigeru Chiba. Programming support of design patterns with compile-time reflection. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, pages 56–60, 1998.
47. Betrand Meyer. *Object-oriented software construction*. Prentice Hall, 2nd edition, 1997.
48. R. Kramer. iContract - the Java(tm) design by contract(tm) tool. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295. IEEE Computer Society, 1998.
49. Barbara Liskov. *CLU Reference Manual*. Springer-Verlag New York, Inc., 1983.
50. Stephan Murer, Stephen Omohundro, David Stoutamire, and Clemens Szyperski. Iteration abstraction in Sather. *ACM Transactions on Programming Languages and Systems*, 18(1):1–15, 1996.
51. Free Software Foundation. GNU compiler collection. `http://www.gnu.org/software/gcc/`, 2003.
52. STLlint: Static checking for the C++ STL. `http://www.cs.rpi.edu/~gregod/STLlint`.
53. Christian Bauer, Alexander Frink, and Richard Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *Journal of Symbolic Computation*, 33(1):1–12, 2002.
54. Semple static analysis engine. `http://www.cs.rpi.edu/~gregod/Semple`.

APPENDIX

A. Example Iterator Specification

```
template<typename Iterator, typename Sequence>
  struct safe_iterator : iterator_traits<Iterator>
  {
    safe_iterator(unsigned int pos, const Sequence* seq)
      : sequence_(seq), version_(seq->version_), position_(pos) { }

    safe_iterator(const safe_iterator& other)
      : sequence_(other.seq), version_(other.version_), position_(other.position_)
    { semple_assert(!singular(), "attempt to copy a singular iterator"); }

    typename safe_iterator::reference operator*() const {
      semple_assert(deferenceable(),
                    "attempt to deference an iterator that is not dereferenceable");
      return sequence_->data_;
    }

    safe_iterator& operator++() {
      semple_assert(incrementable(),
                  "attempt to increment an iterator that is not incrementable");
      ++position_;
      return *this;
    }

    bool operator==(const safe_iterator& other) const {
      semple_assert(!singular() && !other.singular(),
                    "attempt to compare a singular iterator");
      return position_ == other.position_;
    }

    bool operator!=(const safe_iterator& other) const { return !(*this == other); }

    bool singular() const { return !sequence_ || sequence_->version_ != version_; }

    bool dereferenceable() const
      { return (sequence_ && position_ >= 0 && position_ < sequence_->size_); }

    bool past_the_end() const { return position_ >= sequence_->size_; }

    bool incrementable() const { return dereferenceable(); }

    const Sequence*  sequence_;
    unsigned int     version_;
    unsigned int     position_;
  };
```

B. Example Vector Specification

```
template<typename T>
  class vector
  {
  public:
    typedef safe_iterator<T*, vector<T> > iterator;

    vector() : size_(0), version_(1), min_capacity_(0) { }
    iterator begin() { return iterator(0, this); }
    iterator end() { return iterator(size_, this); }

    unsigned int capacity() const {
      unsigned int x; // uninitialized
      semple_assume(x >= min_capacity_);
      return x;
    }

    void push_back(const T& x) {
      if (size_ == 0 || random()) data_ = x;
      if (size_ >= min_capacity_) { ++version_; ++min_capacity_; }
      ++size_;
    }
    unsigned int size_;
    unsigned int version_;
    unsigned int min_capacity_;
    T            data_;
  };
```