HIGH-LEVEL STATIC ANALYSIS FOR GENERIC LIBRARIES

By

Douglas Gregor

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved by the
Examining Committee:

____
Sibylle Schupp, Thesis Adviser

____
John Field, Member

____
Mukkai Krishnamoorthy, Member

____
Ana Milanova, Member

____
David Musser, Member

Rensselaer Polytechnic Institute
Troy, New York

April 2004
(For Graduation May 2004)

HIGH-LEVEL STATIC ANALYSIS FOR GENERIC LIBRARIES

By

Douglas Gregor

An Abstract of a Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

The original of the complete thesis is on file
in the Rensselaer Polytechnic Institute Library

Examining Committee:

Sibylle Schupp, Thesis Adviser

John Field, Member

Mukkai Krishnamoorthy, Member

Ana Milanova, Member

David Musser, Member

Rensselaer Polytechnic Institute
Troy, New York

April 2004
(For Graduation May 2004)

© Copyright 2004
by
Douglas Gregor
All Rights Reserved

CONTENTS

LIST OF TABLES . . . viii

LIST OF FIGURES . . . ix

ACKNOWLEDGMENT . . . xii

ABSTRACT . . . xiii

1. Introduction . . . 1
   - 1.1 Static analysis . . . 2
     - 1.1.1 Language-level static analysis . . . 3
     - 1.1.2 Library-level static analysis . . . 4
     - 1.1.3 Static checking . . . 6
   - 1.2 Problems and challenges . . . 7
     - 1.2.1 Analyzing abstractions . . . 7
     - 1.2.2 High-level vocabulary, low-level language . . . 9
     - 1.2.3 Higher-level iteration constructs . . . 9
     - 1.2.4 Managing extensibility . . . 11
   - 1.3 Solutions . . . 12
     - 1.3.1 Checking against specifications . . . 12
     - 1.3.2 Object-aware static analysis . . . 13
     - 1.3.3 High-level loop analysis . . . 14
     - 1.3.4 Multiple dynamic classification . . . 14
     - 1.3.5 Algorithm concepts . . . 15
   - 1.4 Contributions . . . 16
   - 1.5 Notation . . . 17
   - 1.6 Outline of the thesis . . . 18

2. Related work . . . 20
   - 2.1 Static analysis . . . 20
     - 2.1.1 Abstract interpretation and symbolic execution . . . 20
     - 2.1.2 Integer analysis . . . 21
     - 2.1.3 Loop analysis . . . 23
     - 2.1.4 Higher-level static analysis . . . 26

2.2 Static checking 27
2.2.1 Language-level checking 27
2.2.2 Specification-based checking 28
2.3 Generic programming 30
2.3.1 Concepts 30
2.3.2 Run-time checking of STL implementations 31
3. Static checking for generic software libraries 33
3.1 Overview of the Standard Template Library 33
3.1.1 Containers 34
3.1.2 Iterators 34
3.1.3 Algorithms 37
3.2 Static checking for the Standard Template Library 38
3.3 High-level static checking methodology 41
3.3.1 Standard Template Library specifications 42
3.3.2 Symbolic execution 43
3.3.3 Program loops 46
3.3.4 Recursion 46
4. The Semple language 50
4.1 Motivation and goals 50
4.2 Syntax and semantics 51
4.2.1 Lexical conventions 51
4.2.2 Declarations 52
4.2.2.1 Classes 52
4.2.2.2 Variables 52
4.2.2.3 Subroutines 52
4.2.3 Types 53
4.2.4 Expressions 53
4.2.4.1 Boolean expressions 53
4.2.4.2 Integer expressions 54
4.2.4.3 Pointer expressions 55
4.2.4.4 Lvalue expressions 55
4.2.4.5 Miscellaneous grammar productions 56
4.2.5 Statements 56
4.2.6 Translation unit 59
4.3 Specification example 59

5. Integer analysis . . . 62
5.1 Lazy symbolic range propagation . . . 63
5.2 Precision loss due to relational overlap . . . 65
5.3 Relational merging algorithm . . . 69
5.4 Variable relation graph . . . 71
5.4.1 Tree properties . . . 72
5.4.2 Graph properties . . . 73
5.4.3 Spanning tree selection and pruning heuristics . . . 74
5.5 Merging objects . . . 76
5.5.1 Static field relation graph . . . 77
5.5.2 Computing the static field relation graph . . . 78
5.5.3 Dynamic field relation graph . . . 79
5.5.4 Instantiation . . . 81
5.5.5 Pruning the dynamic field relation graph . . . 84
5.6 Algorithm complexity . . . 84
5.7 Summary . . . 86
6. Loop analysis . . . 87
6.1 Induction variable recognition . . . 89
6.1.1 Symbolic differencing . . . 89
6.1.2 Trip count calculation . . . 90
6.1.3 Monotonic induction variables . . . 91
6.2 Interprocedural loop analysis . . . 92
6.2.1 Interprocedural trip count calculation . . . 93
6.2.2 Goal-directed inlining . . . 95
6.3 Proving noninductive equivalence . . . 97
6.4 Loop analysis algorithm . . . 99
6.5 Summary . . . 101
7. Organization of an extensible static checker . . . 103
7.1 Algorithm concepts . . . 104
7.2 Customization via events . . . 107
7.3 Implementation of static event dispatch . . . 110
7.4 Summary . . . 112

8. STLlint: an extensible static checker for the STL . . . 113
8.1 Scope . . . 113
8.2 Examples . . . 114
8.2.1 Iterator invalidation . . . 114
8.2.2 Writing to a past-the-end iterator . . . 115
8.2.3 Inefficient algorithm selection . . . 118
8.3 Design and implementation . . . 118
8.3.1 Static checking primitives . . . 120
8.3.2 C++ standard library specifications . . . 121
8.3.3 Implementation . . . 123
8.4 Experimental results . . . 124
8.4.1 Precision . . . 124
8.4.2 Efficiency . . . 126
8.5 Additional capabilities . . . 129
8.5.1 Library-centric diagnostics . . . 129
8.5.2 Generic algorithm safety checking . . . 132
8.6 Limitations . . . 133
8.7 Summary . . . 134
9. Conclusions . . . 135
10. Future work . . . 136
10.1 Integration of static and run-time checking . . . 136
10.2 Fragment analysis . . . 137
10.3 Other iteration models . . . 137
REFERENCES . . . 139
APPENDICES . . . 149
A. Example generic iterator specification . . . 149
A.1 Support macros . . . 151
A.2 Construction and assignment . . . 151
A.3 Dereference operations . . . 153
A.4 Input iterator operations . . . 154
A.5 Bidirectional iterator operations . . . 155

A.6 Random access iterator operations 156
A.7 Iterator invalidation 158
A.8 Iterator comparison operators 158
B. Example container specification: STL `vector` 160
B.1 Safe sequence base class 162
B.1.1 Iterator invalidation 163
B.2 Construction and destruction 163
B.3 Assignment 164
B.4 Capacity 165
B.5 Modifiers 166
C. Experimental Results 170

LIST OF TABLES

C.1 STLlint's execution time on the STLlint test suite . . . 170

C.2 Static measures of Semple program size and complexity in the STLlint test suite . . . 174

C.3 Number of integer operations performed by STLlint relative to the analysis time . . . 178

LIST OF FIGURES

1.1 Specification of the `sort_heap` algorithm excerpted from the C++ standard [5, §25.3.6.4] 5

1.2 An example of the "STL style" of C++ programming, where algorithms operate on data structures only through the iterator abstraction 10

3.1 Implementation of the STL `find_if` algorithm 35

3.2 The lattice of STL iterator concepts 36

3.3 [`sorted_vec_insert.cpp`] The iterator `lb` may be invalidated by the call to insert, but it only occurs when the size of the `vector` equals its capacity on entry. 38

3.4 [`insert_other.cpp`] The call to `insert` is ill-formed because the iterator `i` refers to the wrong container. 39

3.5 [`erase_remove.cpp`] The invocation of the `remove` algorithm reshuffles elements, but does not actually remove them, so the `vector v` contains some elements with undefined value. 40

3.6 Partial call graph for the program snippet in Fig. 3.3. User code is represented by empty nodes, library implementation code is represented by shaded nodes, and the remaining nodes represent interfaces between library and user code. 41

3.7 A simple program to compute an overall grade 44

3.8 Trivial join lattice used to construct a symbolic domain from a concrete domain that consists of values v_1, v_2, ..., v_n 45

3.9 Verification of a subroutine that erases students with a failing grade from a vector requires precise loop analysis. 47

3.10 Interprocedural control-flow graph for the recursive formulation of the Fibonacci function. 48

4.1 Semple specifications of a `vector` and its `iterator` type 59

4.2 Specification of the `vector insert` member function excerpted from the C++ standard [5, §23.4.2.3] 60

4.3 Semple specification of the `vector insert` function 61

5.1 The range dependency graph for the expression $d = 1 + (x \in [0 : N-1]) - N$ 63

5.2 Example code illustrating one case where lazy symbolic range propagation is more accurate than the original formulation by Blume and Eigenmann 64

5.3 Merging the value ranges representing the fields in an interval abstract data type results in an unacceptable loss of precision. 67

5.4 A depth-first search visitor that performs relative merges along a spanning tree in the variable relation graph 72

5.5 A variable relation tree where vertex v_1 is arbitrarily selected as the root 73

5.6 The transitive reduction of a directed variable relation graph may reduce the effective precision of the merge operation. 75

5.7 Static field relation graphs for the specification of the interval, iterator, and vector abstract data types 77

5.8 Dynamic field relation graph for the objects in Fig. 3.9 80

5.9 Points-to graph for the loop in Fig. 3.9 83

6.1 A simple loop containing a basic induction variable i and two monotonic induction variables, j and k 91

6.2 Partial expansion of the `find_if` function from Fig. 3.1, illustrating the static analyzer's view of the "simple" iterator loop 94

6.3 "Optimized" form of the `do`–`while` loop from Fig. 6.2, after boolean variable definitions have been "pulled" into the `if` condition 96

6.4 A loop containing two noninductive but equivalent values, `i->version` and `i->v->version` 98

7.1 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, because it relies on the default `<` ordering. 104

7.2 Partial algorithm concept lattice describing Standard Template Library algorithms 106

7.3 C++ representation of the algorithm concept lattice in Fig. 7.2 108

7.4 Skeletal implementation of the `lower_bound` algorithm including user annotations associating the implementation with the LOWER_BOUND concept 109

7.5 Implementation of the `on_entry` event handler 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`. 111

8.1 [`fails_iters_bad.cc`] A misguided optimization of the example code in Fig. 3.9, which results in a comparison against the singular iterator `end_iter` 114

8.2 [`meyers_scary.cpp`] An example adapted from Meyers [72] that executes predictably under unchecked STL implementations, but violates STL requirements 116

8.3 [`sort_slow_search.cpp`] Illustration of inefficient algorithm selection in a user program. STLlint will suggest an algorithmic optimization based on `lower_bound`. 118

8.4 Overview of the STLlint static checker. 119

8.5 Multiple dynamic classification primitives provided by STLlint 120

8.6 Definition of the `_M_invalidate_from` method, used by the STLlint `vector` specification, that invalidates all iterators referencing the current container whose positions equal or exceed the input value `__n` . . . 121

8.7 STLlint is unable to deduce that the values of `i` across all iterations are distinct, and therefore conservatively determines that the resulting size of the container `v` varies falls in the range [0 : 20]. 125

8.8 STLlint is unable to verify the correctness of the second loop due to the insufficient modeling of the stable `multiset` iterators. 126

8.9 Histograms illustrating the analysis time (in seconds) required by STLlint and the ratio of analysis time to front end time for our STLlint test suite127

8.10 Plots illustrating the relationships between analysis time and the number of integer comparisons or variable replacements performed by lazy range propagation 130

8.11 Diagnostic output produced by STLlint that illustrates error message grouping and linking to external documentation 131

ACKNOWLEDGMENT

First and foremost, I would like to acknowledge the great impact of my advisor and friend, Sibylle Schupp, on my research, this thesis, and on me personally. Her near-infinite patience saw all manner of wayward ideas, unworkable schemes, and frivolous diversions, yet her subtle guidance shaped them into a coherent path. Coupled with her unbounded enthusiasm and keen insight, I could not imagine a better guide through the trials of doctoral research and the thesis-writing process.

I would also like to thank David R. Musser for the many interesting discussions on generic programming, formal verification, and myriad other topics that will continue to influence me for years to come. His contributions to the field of generic programming and to this research have been invaluable.

I am forever grateful to my parents, who instilled in me an unquenchable thirst for knowledge and an insatiable desire for learning and discovery. They taught me that no problem is too complex and no answer beyond reach; that with a little luck and a little patience any achievement is possible.

Lastly, I thank my wife Amy for her unwaivering love and support throughout this process. Only the wife of a doctoral student can comprehend the horrors of late-night lectures that none should be forced to endure, yet she shared in my excitement and in my dream.

ABSTRACT

Software developers are increasingly reliant on well-tested, reusable software libraries that can rapidly decrease development time by eliminating the need to reimplement standard algorithms and data structures. Generic software libraries in particular, such as the C++ Standard Template Library (STL), provide efficient, reusable solutions that can be customized for nearly any computing environment.

Generic software libraries introduce a large number of abstractions, such as subroutines and classes, which simplify the task of a software developer. These abstractions, however, complicate static analyses, causing traditional static analysis techniques to produce poor, imprecise results. In contrast, a higher-level static analysis exploits these same abstractions to produce precise program analysis information at the library's level of abstraction, enabling library-level optimization and verification.

This thesis introduces a higher-level static analysis that employs executable specifications of software libraries to reduce the complexities of library analysis and improve analysis precision. Moreover, our analysis exploits object-oriented abstractions to improve the precision of analysis for a very important class of iterator movement problems. Other abstractions, such as subroutine abstractions and pointers, are effectively handled with symbolic analysis methods, even in program loops with nontraditional induction variables.

This thesis also introduces an organization methodology for extensible specifications for generic software libraries based on algorithm concepts. This methodology minimizes the effort required to introduce new algorithms, data structures, and semantic properties into generic library specifications.

The methodologies and analyses presented are implemented in the STLlint static checker. STLlint is able to detect errors in the use of the Standard Template Library within C++ user programs, including semantic checks on the abstract properties of STL algorithms. Our experiments have shown that STLlint achieves a low false positive rate and is capable of diagnosing common STL usage errors.

CHAPTER 1
Introduction

Program analysis is the study of algorithms for automatic deduction of the behavior and properties of computer programs. The information discerned by program analysis techniques can be applied to perform or aid in various software engineering tasks, such as transformation of programs between languages, optimization of programs, verification of program correctness or safety, debugging, or computation of various measures of software quality.

Programming languages are the primary method of communication with development tools. They evolve to support newer programming methodologies, but the evolution of mainstream programming languages is very slow. To circumvent this slow process, programmers mimic newer methodologies within older, established languages. For instance, design patterns [37] are carefully applied by mimicking prototype implementations and object-oriented programming is applied to lower-level languages such as C by careful organization of data structures. More commonly, functionality that would naturally be presented as a domain-specific language, such as linear algebra computations, is instead provided as a software library.

Software libraries provide capabilities far beyond what a language can—or should—supply, offering in essence domain-specific sublanguages within a general-purpose language. The C++ programming language [5] in particular provides extensive (albeit accidental) support for the construction of software libraries that can accurately represent domain-specific syntax and semantics. The C++ Standard Template Library [81, 106], for instance, provides generic, reusable data structures and algorithms for search and sorting applicable to many different domains, but other software libraries have addressed areas as varied as scientific computing [86, 103, 113], compiler technology [23], and computational theory [33, 53, 104].

Although software libraries integrate seamlessly with the development process, they do not enjoy the same benefits that would be provided to a domain-specific language, such as domain-specific optimization or verification. Compilers do not

optimize based on library semantics, debuggers do not present library components as cohesive entities, and program verifiers will report errors in the use of the library only when an error is detected in the library implementation. The result is that programmers must understand both the abstractions present in the software library and their implementation within the general-purpose language. Thus, programmers would benefit from development tools that better support libraries, by treating them as true sublanguages in the general purpose language and alleviating the need to understand the implementation of the library.

To support software libraries, program analyses must not be limited by the lower-level semantics of the library implementation. Analysis of the unique high-level semantics of software libraries requires special techniques not employed by existing program analyses; these analyses thus fail to provide information that is adequate for various software engineering tasks. This thesis introduces the notion of *library-level* program analysis and illustrates the challenges inherent to *lifting* program analysis methods from the level of languages to the level of libraries. To address these unique challenges, this thesis presents new program analysis methods that effectively cope with high-level constructs found in generic software libraries. Certain programming methodologies, especially those that employ generic software libraries such as the Standard Template Library, rely greatly on high-level library constructs that benefit from the library-level analyses described in this thesis.

The following sections describe the difference between language-level and library-level static analysis and the challenges one faces when lifting an analysis to a library-level analysis, outline the solutions proposed in this thesis to address these challenges, and enumerate the specific research contributions detailed in this thesis.

1.1 Static analysis

Static analysis is a form of program analysis that seeks to discover properties of a program's behavior by analyzing a representation of the program's source code. Static analysis is the most common form of analysis, and is typically applied in compiler optimizations, documentation generation systems, and offline program verification.

Static analysis is characterized by the need to consider all valid executions of a given program, independent of knowledge of any particular input. Thus all static analysis techniques are necessarily *conservative* in their approximations of program state. The goal of a static analysis is therefore to provide adequate *precision*, i.e., a close approximation to the actual set of program states that may be achieved, while retaining reasonable efficiency.

Static analyses are often characterized by their behavior with respect to certain language features. An *intraprocedural* analysis considers only the operations within a single subroutine, whereas an *interprocedural* analysis considers the effects of other subroutines. An interprocedural analysis may be *context-sensitive*, indicating that each call to a particular subroutine will be analyzed separately. On the other hand, a *context-insensitive* analysis will analyze each subroutine only once, approximating all inputs and outputs to that subroutine. *Flow sensitivity* implies that the analysis considers the order of execution of program statements; a *flow-insensitive* analysis assumes that the statements can be executed in any order. Finally, a *path-sensitive* analysis involves analyzing each path through the program (marked by the decisions made at each conditional branch) separately; a *path-insensitive* analysis merges the results from separate paths together.

1.1.1 Language-level static analysis

The vast majority of static analyses are language-level analyses, meaning that they analyze program source code relative to the semantics of the programming language in which it is written. The abstraction level of a language-level static analysis is necessarily tied to the abstraction level of the language, and can only be lowered from there, with each lowering step eliminating some amount of higher-level structure. For instance, a static analysis for an assembly language will derive information about programs relative to the machine registers and the state of the memory, but will not deduce the types of parameters passed to a function, since this information is not present in this representation.

Higher-level languages permit static analyses that deduce more abstract semantic information. For example, static analyses for object-oriented languages can

determine the properties of relationships amongst objects and static analyses for a linear algebra language could deduce that, for instance, a particular matrix is a diagonal matrix. With higher-level information, higher-level transformations can be applied, e.g., eliminating multiplication by the identity matrix or replacing the multiplication of two diagonal matrices by elementwise multiplication of their diagonals.

The analysis of higher-level languages is often different than the analysis of lower-level languages, owing to the need to perform different types of analysis for different kinds of entities. For instance, role analysis [65] seeks to verify the relationships amongst objects in object-oriented programs. Higher-level abstractions in languages can even simplify analysis by providing additional structural information, e.g., object-oriented programming allows us to distinguish between the implicit object parameter ("`this`") and other pointer parameters to a subroutine, which can be exploited to improve analysis, e.g., by providing context sensitivity for only the implicit object parameter [73].

1.1.2 Library-level static analysis

Libraries are a compromise between the benefits of higher-level, domain-specific languages and lower-level, general-purpose languages, because they provide a high-level veneer over a general-purpose language. Library-level static analysis treats this veneer as a sublanguage, to enable higher-level static analyses for libraries within the context of a general-purpose language.

Library-level static analysis requires that one give library types and operations real semantics, beyond the semantics they intrinsically acquire from their implementations. For instance, a class implementing an integer type of unbound length is no longer just a collection of unrelated bits; it is an algebraic type with invariants and a set of operations interrelated by mathematical axioms.

Library semantics vary greatly from one library to the next, and are often far more complicated than the semantics of general-purpose programming languages. Fig. 1.1 contains a specification from the C++ Standard Template Library [81, 106] (part of the ANSI/ISO C++ standard [5]). This specification illustrates how the

```
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 $N = \texttt{last} - \texttt{first}$).
3. **Notes:** Not stable.

Figure 1.1: Specification of the `sort_heap` algorithm excerpted from the C++ standard [5, §25.3.6.4]

problem of analyzing libraries is markedly different from that of analyzing low-level languages. The *effects* clause describes several preconditions and postconditions [48], including:

- The parameters `first` and `last` are iterators [77] that must form a *valid range*. Iterators abstract the notion of iteration over a sequence of values, such as values within a container, and will be discussed further in Sec. 3.1.2.
- The values in the sequence referenced by `[first, last)` must be arranged as a heap. It is specified elsewhere [5, §25.3] 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 in the sequence `[first, last)` into sorted order, based on the ordering relation `comp` or the $<$ operator.

We note that these pre- and postconditions are very high-level semantic properties. The notion of a valid range is based on the iterator *concept* [8, 76], i.e., a set of syntactic and semantic requirements coupled with a set of abstractions that fulfill these requirements [57]. Thus to verify that the parameters `first` and `last` form a valid range, we aren't verifying requirements based on language constructs (the

C++ *language* does not have iterators), or even based on a concrete type or interface, but are verifying requirements specified for an infinite (and unknown) set of data types. Moreover, the "heap" precondition and "sorted" postcondition describe particular orderings on the values that will be referenced by the given iterators. Overall, a static analysis given such a high-level specification must cope with abstractions far from the language itself, to user-defined data types and even to infinite sets of unknown data types described only by their conceptual requirements.

In contrast, the cost of embedding domain-specific languages into other languages via libraries is that any semantics present in libraries are completely lost, as the libraries are ignored or replaced with their implementations. This results in lost optimization opportunities [44, 95, 96] and poorer run-time performance due to the so-called abstraction penalty [50, 74, 87], poor diagnostics when type errors are detected within C++ generic libraries [101, 119, 122], and the need for programmers to completely understand the abstractions provided by the library along with the implementation of those abstractions in the language.

The goal of library-level static analysis is to provide the same opportunities for optimization, verification, and discovery of programs making use of software libraries that would be available if those libraries were instead implemented as domain-specific languages. Library-level static analyses are extensible to many different kinds of software libraries, and may require that the libraries assist the analysis process. Such libraries are called *active libraries* [112, 114], because they take an active role in their own analysis and optimization.

This thesis presents static analysis techniques capable of producing precise, high-level information for programs that use high-level libraries such as the Standard Template Library. The static analyses described are implemented in the STLlint static checker, which is able to detect bugs in programs using the STL, including violations of the high-level specification illustrated by Fig. 1.1.

1.1.3 Static checking

Static checking is the application of static analysis to detect bugs in program code. Static checking is a form of verification that is typically limited to uncovering

safety violations (e.g., errors that would crash a program) and security violations (e.g., errors that would allow unauthorized access to information through the program), whereas one typically applies the term *formal verification* to the process of determining whether a given program conforms to its specification.

The best known static checker is the venerable `lint` [56] checker for the C language, which detects, among other things, nonportable type casting and uses of uninitialized variables. There are many different static checkers available, each of which detects different kinds of program bugs by analyzing the program's source code. The vast majority of these are language-level static checkers (that use language-level static analysis), and we will refer to these static checkers collectively as `lint`-like checkers.

Library-level static checking, on the other hand, involves finding bugs in the use of arbitrary software libraries. For instance, reading from a file or network socket before it has been opened or connected is a common error that might be detected by a library-level checker. A library-level checker might apply a formal version of the specification in Fig. 1.1 to verify proper use of the `sort_heap` algorithm, and therefore needs to be much more open and extensible than a `lint`-like checker. Library-level static checking has also been called the component-client conformance problem [85].

This thesis focuses on static analysis for the purpose of static checking. Analysis is the first step toward optimization, although we do not discuss the issue of high-level optimization, and static checking provides a direct way to evaluate the results, in particular the precision, of an analysis.

1.2 Problems and challenges

This section details the problems and challenges inherent to the process of lifting existing language-level static analysis to the level of software libraries.

1.2.1 Analyzing abstractions

The primary challenge of high-level static analysis is coping with and exploiting abstractions present in program code. Abstractions include both language-level

abstractions, such as structured data, subroutines, and objects; and higher-level abstractions not directly supported by the language, such as object relationships, concepts, and semantic properties. The challenges presented by language-level and higher-level abstractions differ greatly, as does the effectiveness of various program analyses.

Language-level abstractions present a barrier for many existing static analysis techniques. Static analyses are often *intraprocedural*, meaning that they operate only within a given subroutine, or are *context-insensitive*, meaning that they do not distinguish between two different calls to the same subroutine. While these properties are not a great hindrance for low-level languages, such as a three address code representation in a compiler's back end, they have a drastic effect on higher-level languages that make heavy use of subroutines. For instance, after replacing the integer addition $x + 0$ with a subroutine call $add(x, 0)$, the simple analysis required for optimizing $x + 0$ to x no longer suffices: we need to either eliminate the subroutine abstraction via inlining, or use an *interprocedural*, *context-sensitive* analysis to enable the same optimization. An interprocedural analysis is required because it must consider the implementation of *add*. A context-sensitive analysis is then required to distinguish between different calls to *add* that occur within the program, some of which may be optimizable. Other language-level abstractions, such as the introduction of structured data and pointers, produce similar complications.

Higher-level abstractions present problems different from those of language-level abstractions, because here we wish to operate with semantic properties not present in the language. Revisiting the analysis of $add(x, 0)$, we consider the analysis of the expression when x is an object of a library-defined type, such as an arbitrary-precision integer. In this case, we can no longer rely on the built-in + operator, because the *add* function operates on arbitrary-precision integers. To determine the equivalence of $add(x, 0)$ with x, the static analysis must either deduce the equivalence given the implementation of the *add* function, which is often undecidable, or it must be provided with ample semantic information describing the *add* function.

Both types of abstractions complicate static analysis, as an analyzer must see through the language-level abstractions that thwart many common analyses and

reason with the higher-level abstractions required for precise analysis of software libraries.

1.2.2 High-level vocabulary, low-level language

The vocabulary of specifications for libraries, as shown in Fig. 1.1, is very high-level, although the implementation of these specifications is low-level, at the level of the language. A higher-level static analysis must process the high-level vocabulary, perform analysis on the low-level representation of the program using the high-level semantics, and map the result back into the high-level vocabulary. The inputs and outputs of a high-level analysis are restricted to the high-level vocabulary of the language to accommodate users of libraries and optimizations based on library semantics. Analysis is performed on a lower-level representation so that programs using language abstractions, such as integers and pointers, can be analyzed at the appropriate abstraction level.

Within the Standard Template Library (STL), we are concerned with "iterators," and we often talk about "dereferenceable," "past-the-end," and "singular" values for iterators, along with "valid range" relationships between iterators, and referencing relationships between iterators and containers (all of which will be discussed in Sec. 3.1.2). These concepts have no counterpart in C++, although there are many specific examples of these abstractions that are implemented within the language. A higher-level static checker must analyze C++ code that uses iterators based on the semantics of iterators, and must present any bugs found in the terminology of iterators, not the terminology of any particular implementation of iterators.

1.2.3 Higher-level iteration constructs

The iterator abstraction, pioneered by the Standard Template Library [81, 106], enables data structures and algorithms to be implemented separately, such that any given algorithm can be applied to many different data structures. Iterators are conceptually similar to pointers, in that they can reference the elements in a particular container and are able to move throughout the elements in that container. Unlike pointers, however, the underlying representation of the containers is

```
vector<Student_info>
extract_fails(vector<Student_info>& students)
{
  vector<Student_info>::iterator iter =
    stable_partition(students.begin(), students.end(), pgrade);
  vector<Student_info> fail(iter, students.end());
  students.erase(iter, students.end());

  return fail;
}
```

Figure 1.2: An example of the "STL style" of C++ programming, where algorithms operate on data structures only through the iterator abstraction

unspecified, permitting algorithms to operate on iterators without consideration to the data structure implementation.

The STL has fundamentally changed the way in which C++ programs are developed. The so-called "STL style" of programming focuses on the use of STL data structures and algorithms, employing iterators to communicate with algorithms and eliminate the need for data-structure–specific implementations. Fig. 1.2 illustrates this style of programming; it has been excerpted from a textbook advocating this style for the novice programmer [62]. It employs the `stable_partition` algorithm [5, §25.2.12] to separate students with passing grades from students with failing grades, constructs a new data structure with all of the students with failing grades and erases those students from the `students` data structure. Note that `iter`, `students.begin()`, and `students.end()` all refer to iterators, which pervade this example.

Each of the algorithms used in Fig. 1.2 is implemented via program loops. Unlike most loops written by programmers in other languages and styles, loops written in the STL style are governed by the movements of iterators. Current static analysis techniques can cope with the values of integer variables, integer expressions, some pointer arithmetic (as in C/C++), and sometimes even movement of a pointer variable through a singly-linked list. However, iterators are higher-level iteration constructs whose implementations are potentially very complex, and there is no existing analysis that addresses the movement of STL iterators within loops.

The STL style, which stresses the use of iterators in the application of STL data structures and algorithms, requires a different approach to program loop analysis. This form of analysis, that copes with higher-level iteration constructs, is the greatest challenge of higher-level static analysis. Conversely, without a suitable loop analysis, programs written in the STL style cannot be meaningfully analyzed.

1.2.4 Managing extensibility

Software libraries are designed to be reusable. Generic software libraries—such as the STL—go much further, allowing nearly any part of the library to be customized or extended. Thus any attempt at specifying the semantics of a library must describe this semantics as a function of the semantics of the participants, some of which may come from the library and others that come from user code. The specification of `sort_heap` in Fig. 1.1 is very open, using concept specifications and informal language to convey meaning without resorting to a specific class of implementations. The challenge of managing extensibility is that we must transform these specifications into formal language without tying them to a particular set of implementations, because the set of implementations is infinite.

The extensibility of formal library specifications must involve extensibility to new libraries, new library components (e.g., new data types or subroutines), aggregation and composition of components from other software libraries and user code (e.g., the specification for a container data type must aggregate well with a specification for the element type stored in the container), and new semantic properties (e.g., checking that binary searches are performed only on sequences of elements that have been sorted).

The complexity of managing extensibility is in minimizing the amount of work required to introduce a new semantic property or new subroutine. We evaluate solutions for extensible library specifications by the amount of specification or implementation code that must be (re-)examined to introduce new specifications or semantic checks. Poor extensibility is marked by the need to reexamine all specifications within a library (or other libraries!) to introduce a single new entity.

1.3 Solutions

This section outlines the solutions presented by this thesis to address the unique challenges inherent to higher-level static analysis. While each of these solutions is useful by itself, it is the combination of these techniques that enables precise high-level static checking for programs using the C++ STL.

To permit implementation-independent analysis and express the high-level semantics of libraries, we adopt a specification-based approach to analysis. The specifications, particularly those of iterators, are crafted with an explicit emphasis on constructs suitable for loop analysis, such as integer values. We therefore focus on the analysis of integer values, particularly their behavior within program loops, and construct a new *symbolic* integer analysis that can cope with language-level abstractions such as subroutine calls and pointers. This analysis exploits relationships among program objects (e.g., iterators and containers) to improve the precision of the results computed by our program analysis.

While higher-level iteration constructs permeate programs written in the STL style, other higher-level semantic properties introduced by STL algorithms can greatly affect program correctness and safety. High-level analysis permits the specification and verification of these properties by way of multiple dynamic classification [11], which can extend existing objects with new types, and an algorithm concept [80, 97] taxonomy that reduces the burden of specifying the semantics of every library entity with respect to every semantic property. The following subsections summarize the major design decisions.

1.3.1 Checking against specifications

We view software libraries as consisting of a set of interfaces (a syntax to which clients of those libraries must conform), semantics (the behavior of library components), and implementations (the realization of those semantics). By providing a notation by which a formal semantics of the library can be written, we split the problem of high-level analysis into two distinct problems: checking that a user program conforms to the interface semantics of the library and checking that an implementation in fact implements the stated semantics of the library.

We focus only on the former problem—that of analyzing a user program in the context of a library's semantics—so that we are able to provide formal semantics for libraries without considering concrete implementations. This permits analysis and checking when implementations are unavailable (for instance, because the source code is not provided), several implementations are available (for instance, there are several different implementations of the C++ standard library), or no implementation can exist (for instance, our interfaces are abstract concepts).

Formal specifications of library semantics therefore permit static analysis when it would otherwise be impossible for lack of a single implementation. Moreover, the use of specifications in lieu of implementations allows one to simplify the static analysis: for instance, the iterator abstraction hides complex iterator implementations behind a simple, pointer-like interface. To precisely analyze an iterator implementation for a balanced binary tree with parent pointers (a common implementation for the STL `map` data structure) requires sophisticated shape analysis [40, 91]. On the contrary, a formal specification of an iterator need only consider the interface-level semantics, and can be reduced to the problem of precise pointer arithmetic analysis.

Specifications in this thesis are written in our Semple analysis language, which combines aspects of imperative, object-oriented programming languages with specification primitives tailored to high-level static analysis.

1.3.2 Object-aware static analysis

Relationships among objects (or other forms of abstract data types) defined by software libraries are often important to library semantics. Within the STL, there are ordering relationships among iterators, relationships between iterators and the containers they reference, and relationships between the ordering of values in containers and the algorithms that (indirectly) modify those containers.

This thesis describes a method of representing the relationships among different class types and among objects within a system, through a data structure not unlike the role reference diagram used in role analysis [65]. However, while role analysis [65] is primarily concerned with the specification of object roles and verification that objects retain their roles, this thesis discusses methods to exploit knowledge of

interobject relationships to improve the precision of static analysis, particularly the precision of value range propagation [47] and loop analysis.

1.3.3 High-level loop analysis

Higher-level static analysis is complicated by the use of abstract data types within program loops, especially when these data types govern the behavior of the loops. This situation occurs frequently with iterators, and is addressed by this thesis via high-level loop analysis.

High-level loop analysis is presented here as an aggregate of traditional loop analysis techniques with high-level information about objects and data types. Traditional loop analysis is lifted from a very narrow, low-level, intraprocedural view of the program state to an open, high-level, interprocedural view of the program state that embraces the abstractions present in software libraries. High-level loop analysis involves techniques such as symbolic differencing [46] and symbolic execution [61] to cope with additional abstractions resulting from software libraries, and exploits knowledge of interobject relationships to improve precision where traditional loop analysis techniques fail.

1.3.4 Multiple dynamic classification

Algorithms within software libraries often require their arguments to possess certain transient properties not part of the type system, such as "the input is sorted." Fig. 1.1 illustrates one such algorithm, for which a static checker should verify the precondition that the input sequence is ordered as a heap and assert the postcondition that the sequence is sorted. To do so requires the addition of semantic information to sequence specifications without affecting the sequence specification itself, e.g., the semantic specification for a "sorted" property of a sequence is distinct from the semantic specification of the sequence data structure.

Multiple dynamic classification [11] is an extension for object-oriented languages that allows any particular object to possess multiple types ("classifications") at run time, where classifications may be freely introduced, modified, and removed. The introduction of multiple dynamic classification into our Semple specification

language permits algorithms to extend other specifications by introducing new classifications, which themselves may contain additional data, and query these specifications. For instance, a sequence of values can be classified as "sorted," and may additionally store information about the manner in which the sorting was performed. Multiple dynamic classification therefore enables specifications such as that of Fig. 1.1 to be expressed in a self-contained module.

1.3.5 Algorithm concepts

Concepts used in the STL and its predecessors [76, 77, 106] have been limited to requirements on the operations related to particular abstract data types. Concepts present a minimal interface and semantics by which abstract data types may be classified, providing a medium for separating algorithms from data structures.

Within the context of static analysis for software libraries, a similar classification problem occurs with algorithms. Software libraries often provide several implementation variants of a particular algorithm; the STL, for instance, provides no less than six algorithms that can be used to search for a particular element in a sequence of values. Moreover, from an abstract view even unrelated algorithms often exhibit similar behavior, e.g., an algorithm that randomly shuffles elements in a data structure and an algorithm that sorts elements in a data structure both rearrange the elements but do not insert, remove, or modify any of them.

Algorithm concepts [80, 97] are the natural adaptation of (data type) concepts to algorithm behavior. Like data type concepts, algorithm concepts are a set of requirements and a set of abstractions. The requirements for algorithms, however, are requirements stated on the inputs and outputs of a subroutine, and in our case may involve very high-level semantic assertions like those found in Fig. 1.1.

This thesis will present algorithm concepts as a means to localize semantic descriptions of high-level properties, so that extensions to library semantics can be developed separately, yet interact smoothly. This methodology directly addresses the problem of managing extensibility in an open static checking system for generic software libraries.

1.4 Contributions

This thesis makes the following contributions:

It explores the nature of higher-level static analysis and provides a better understanding of the challenges unique to this form of analysis, particularly those that affect generic libraries in the vein of the C++ Standard Template Library.

Our research provides formal, extensible interface specifications for the containers, iterators, and algorithms of the complete Standard Template Library. These specifications, written in C++ and targeted for our Semple imperative specification language, provide a formal description of the semantics presented in the ANSI/ISO C++ standard [5]. Semple is designed to express the semantic properties of libraries and permit mappings from C++ to enable higher-level program analysis. It includes features specifically tailored to the problems faced in library-level analysis.

This thesis presents a loop analysis method able to cope with high-level iteration constructs, such as the Semple specifications of STL iterators, including those involving multi-level pointer access, higher-level monotonic induction variables, and interprocedural trip count determination. This loop analysis is crucial for analysis of programs that use the Standard Template Library, where many important program loops governed by iterators would typically yield very imprecise analysis results.

Our higher-level loop analysis relies on precise analysis of the values of integer variables and fields in a program. This thesis introduces a new symbolic integer analysis, *lazy range propagation*, and an efficient algorithm that exploits high-level interobject relationships to improve the precision of traditional value range propagation [47] algorithms on which lazy range propagation builds. This algorithm provides a means to retain path-sensitive integral relations across control-flow merges, and is crucial to the precise analysis of iterator idioms, especially those within program loops.

We categorize the properties of STL algorithms into concepts, and arrange these concepts in a concept lattice [120], illustrating commonalities between the algorithms and providing much-needed organization for static analysis on STL algorithms. This lattice enables semantic checks to be decoupled from the specifications of individual algorithms and promotes greater extensibility. The resulting

formal specifications of the STL are easily maintained, extensible by any active library [112, 114], and are able to verify high-level semantic properties.

Finally, this thesis presents the STLlint static checker for C++ programs that use the Standard Template Library. STLlint implements the static analysis methods presented in this thesis, using our Semple analysis language and formal specifications of the STL. It can diagnose many types of errors in the use of the STL, including iterator invalidation, improper use of STL algorithms, and attempts to dereference past-the-end iterators. Our experience has shown STLlint to be effective on test cases derived from freely available STL test suites [38], STL textbooks [8, 62, 72, 81], and errors we have encountered in practice, achieving a 0.59% false positive rate.

1.5 Notation

Throughout this thesis, the discussion will consider C++ and Semple language entities, STL data structures and algorithms, specifications, and several forms of concepts. To distinguish between these entities, the following notational conventions will be used:

- Language entities, including variables, functions, data structures, and algorithms, will be typeset in a `teletype` font. For instance, the function `sort_heap` and the parameter `first` in Fig. 1.1 are both language entities.

- Mathematical entities, such as the (abstract) values of variables will be typeset with an *italics* font, e.g., $x + y \leq z$.

- Particular attributes of specifications, such as the *position* of an iterator or the *size* of a container, will also be typeset with an *italics* font. The different uses of italics will be apparent from context. In many cases, an italicized entity will be both a mathematical expression and a specification attribute, e.g., we will state an invariant such as *position*≥ 0.

- Concepts will be typeset using a SMALL CAPITALS font, to distinguish them from actual types and algorithms. For instance, we may say that the `first` parameter in Fig. 1.1 must model the RANDOMACCESSITERATOR concept.

Some contexts require that several different notational conventions be combined. For instance, we may refer to the *position* attribute of a particular variable, `first`, as `first.`*position*. While somewhat jarring, this notation accurately represents the fact that we are discussing an abstract specification attribute of a variable within a C++ program.

1.6 Outline of the thesis

Chap. 2 presents research related to this thesis, primarily in the areas of static analysis, static checking against specifications, and generic programming.

Chap. 3 provides a review of the C++ Standard Template Library and presents examples of the “STL style” of programming this thesis seeks to address. It then describes our higher-level static checking methodology.

Chap. 4 presents the Semple language. Semple is the internal representation used within the Semple static analysis engine that forms the core of STLlint.

Chap. 5 presents the integer analysis we utilize for higher-level static analysis, which is based on a symbolic form of value range propagation [47] called *symbolic range propagation* [12]. Additionally, this chapter presents the *field relation graph*, a data structure that maps the interobject relationships within a program, which is then used to improve the precision of the merge operation for symbolic value ranges.

Chap. 6 details specific challenges presented by higher-level loop analysis and presents our solution based on symbolic differencing [46]. Specific examples of complex iteration patterns common in “STL style” C++ code but requiring sophisticated loop analysis are presented. This chapter illustrates another use of the aforementioned field relation graph to improve the analysis of noninductive values.

Chap. 7 discusses our algorithm concept lattice for the C++ STL, and in particular its representation and use within STLlint to decouple semantic checking from specific subroutines. We illustrate the encoding of an algorithm concept lattice as a C++ class hierarchy and present an algorithm that uses this hierarchy to execute arbitrary specification code customized for particular algorithm concepts and semantic checks.

Chap. 8 describes the STLlint static checker for C++. This chapter details the

scope and architecture of STLlint, along with the integration of higher-level static analysis techniques into a coherent system. We will also present empirical results gauging the effectiveness of STLlint.

Chap. 9 summarizes this research and Chap. 10 presents potential directions for future work in the area of higher-level static checking.

CHAPTER 2
Related work

This chapter presents a review of existing research in the three areas most related to higher-level static checking: static analysis, static checking against specifications, and generic programming.

2.1 Static analysis

Static analysis refers to the process of determining dynamic properties of programs from their static representation, usually by analyzing some representation of the program source code. This section discusses static analysis techniques related to those presented in this thesis, beginning with underlying frameworks for static analysis and continuing with a review of existing analyses for integer values and program loops. Finally, we discuss analyses that use the abstractions present in high-level languages to improve analysis precision or efficiency.

2.1.1 Abstract interpretation and symbolic execution

Symbolic execution [61] 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 remainder of the program is then executed for each of these program states, in the worst case resulting in a number of program states that is exponential in the number of conditional branches and infinite when faced with certain loops in the control flow graph.

Abstract interpretation [20], like symbolic execution, employs a symbolic representation of the values of program variables to perform static analysis. Abstract interpretation differs in its handling of multiple program states: abstract interpretation computes program states at every program point simultaneously, but performs

program state merging operations at all control-flow joins.

Both methodologies provide powerful frameworks that can easily incorporate analyses of the same type for performance tuning (e.g., two different algorithms for integer analysis) and also allow for simultaneous analysis with completely unrelated analyses (e.g., an integer analysis and a pointer analysis). By intertwining several types of analysis, one can improve the accuracy of each analysis. For instance, pointer analysis may determine that a particular branch will never be taken and therefore integer analysis need not consider that branch; the converse may also occur in the same program, where integer analysis may be able to eliminate a section of code that pointer analysis could not otherwise have ignored. Thus by running these analyses in parallel one can reduce the size of the problem and potentially increase the accuracy of the result by removing two sections of code whereas running the two analyses in sequence would only eliminate one of the sections of code. Therefore, we will favor analyses based on or similar to abstract interpretations in our presentation, although the high-level analysis presented in this thesis is not an abstract interpretation because it does not retain information about all program points and does not apply traditional widening/narrowing operations for loop analysis.

2.1.2 Integer analysis

In the context of this thesis, integer analysis is concerned with approximating the values of integer program variables at each program point. Integer analyses differ mainly in the representation of the integer value approximation, the way in which approximate values are compared, and the function employed to merge two approximations at control-flow joins. The representation of integer values may be as simple as an integral constant (as in constant propagation) or a range of integer values [115]. In some cases the representation is far more complex, involving symbolic ranges [12] or convex polyhedra [21]. The design and complexity of expression comparison and merging algorithms naturally follow the value representation.

Generalized constant propagation [115] utilizes the simplest representation of integer values that we will consider. The value of an integer program variable x is represented by a value range [47] denoted $[a : b]$, where a and b are inte-

gral constants that bound x (i.e., $a \leq x \leq b$). At each control flow graph node, the incoming edges have ranges $[a_1 : b_1], [a_2 : b_2], \ldots, [a_n : b_n]$ for the program variable x, and all of these ranges are conservatively merged into a single range $[\min(a_1, a_2, ..., a_n) : \max(b_1, b_2, ..., b_n)]$. Comparison of expressions within this representation is straightforward: the result of $x < y$ where x falls in the range $[a_x : b_x]$ and y falls in the range $[a_y : b_y]$ is true if $b_x < a_y$, false if $b_y \leq a_x$, and unknown otherwise.

Value range propagation [47] describes a family of analyses. The most expressive formulation uses a slightly more complex representation of value ranges than has been presented, where the bounds in a range are represented by an integer offset from another variable. This representation enables understanding of simple linear relationships between the values of variables, such as "the value of x is one greater than the value of variable y", and can be applied to very simple checking of, e.g., container and iterator usage. Comparing expressions with this representation reduces to solving simple linear systems.

Cousot and Halbwachs [21] propose a much more complex representation of variable constraints based on convex polyhedra in an n-dimensional space of variable values. At conditional branches, new constraints are introduced by bounding the polyhedron according to the test condition. Expression comparison is performed by constructing the hyperplane corresponding to the difference of the involved expressions, and determining where the polyhedron lies with respect to the hyperplane.

Symbolic range propagation [12] represents the bounds of value ranges via symbolic expressions, where each symbol corresponds to a program variable that may itself have symbolic bounds. Symbolic ranges are simplified by a small set of conservative rewrite rules, thereby reducing the size of the symbolic terms to be manipulated. Comparisons in symbolic range propagation are performed by computing the difference between two expressions and replacing variables with their (symbolic) bounds as dictated by the Range Dependency Graph (RDG) [12]. The RDG models the use of symbolic variables within the bounds of other symbolic variables, such that the topological ordering of nodes in the RDG provides a natural replacement order. Like many other value range analyses, its merge operation computes a value

range $[\min(a_1, a_2, ..., a_n) : \max(b_1, b_2, ..., b_n)]$. While more powerful than the analyses previously presented, its computational complexity lies between simple value range propagation [47] and the method of Cousot and Halbwachs [21], making it a prime candidate for use in a precise static analysis system. Symbolic range propagation is detailed in Sec. 5.1, where it is further refined to our lazy range propagation algorithm suitable for higher-level static analysis.

The Value Name Graph (VNG) of Bodik and Anik [13] represents the path-sensitive equality relationships among the values of program variables without the exponential explosion in the state space typical of naïve path-sensitive algorithms. The VNG is a graph describing the relationships between the values of variables at all points within a routine, and while it is very different from the abstract interpretation-based analyses previously presented, it shares a common goal in that it seeks to preserve relationships amongst integer values that many path-insensitive analyses fail to preserve. Sec. 5.2 further describes this problem in the context of value range propagation.

2.1.3 Loop analysis

Loop analysis is a large area of research that can refer to many different types of analysis on program loops. We limit our discussion of loop analysis to two topics of interest for this thesis: induction variable recognition and loop trip count estimation. Induction variable recognition refers to the process of deducing a closed-form expression describing the value of a program variable based only on the initial conditions at the loop preheader and the iteration number. Loop trip count estimation is concerned with statically computing or bounding the number of loop iterations that will be executed at run time.

Traditional loop analyses often employ ad hoc pattern-based techniques to find induction variables. For instance, many compilers recognize *basic* induction variables whose only modification within a loop is of the form `i := i op c`, where `i` is an integer variable, `op` is integer additional or subtraction, and `c` is an integral constant (more advanced techniques allow `c` to be a loop-invariant expression, i.e., an expression whose value is the same throughout all loop iterations). *Generalized*

induction variables are those whose only modification within a loop is of the form `j := a*i + c`, where `i` is a basic induction variable and both `a` and `c` are integral constants (possibly extended to loop-invariant expressions).

Traditional, pattern-based techniques do not extend well to loop analysis problems that occur with iterators and other higher-level induction variables, particularly those modified through pointer indirection and by subroutine calls, because they are limited to the analysis of local integer variables. We defer full discussion of the challenges presented by higher-level induction variables to Chap. 6, but note here that these traditional techniques, even those based on Static Single Assignment (SSA) [4, 88, 121], cannot detect these induction variables.

On the other hand, loop analyses based on the symbolic values of program variables naturally handle subroutines and pointers (given a suitable pointer analysis), because the symbolic values are available independent of the number of times or manner in which they are manipulated. Symbolic loop analyses are built from symbolic integer analyses and therefore integrate well within a symbolic execution framework.

Value range analyses based on abstract interpretation that store program variable values in integral ranges often use *widening* [12, 15, 21, 115] to analyze loops. Widening increases the value range of program variables modified in a loop to provide a conservative approximation of the values throughout the loop's execution, and is performed on back edges to a loop header. The method by which the range is widened varies, but the worst-case termination condition (a lower bound of $-\infty$, upper bound of ∞, or both) applies to all methods. Narrowing [12, 15] is a complementary step to widening that recovers some information lost to widening at loop headers by propagating information gleaned from the loop termination condition across edges flowing from that conditional branch. However, narrowing cannot improve the precision of value ranges for program variables not involved in the loop termination condition, and is therefore less suitable for higher-level induction variables.

Symbolic differencing [46] is a powerful loop analysis technique that integrates well with symbolic range propagation. Unlike widening, symbolic differencing rec-

ognizes induction expressions in the traditional sense. Symbolic differencing uses Newton's forward formula for interpolation to recognize generalized induction expressions of the form $\chi(n) = \varphi(n) + ar^n$ for a polynomial φ of degree m (with loop-invariant coefficients) and loop-invariant a and r. The maximum degree m of φ is a parameter to the loop analysis. Unlike traditional, pattern-based techniques, symbolic differencing naturally copes with the use of pointers, multiple assignments, and subroutines. Chap. 6 describes our application of symbolic differencing to higher-level loop analysis.

Van Engelen presents a novel technique [108] for loop analysis based on chains of recurrences. The technique is safer than symbolic differencing, in that it will not derive an incorrect induction expression when a higher-order induction variable is improperly approximated by a lower-order polynomial. It is also more powerful, in that it recognizes a larger class of induction variables. However, it does not exhibit the same synergy with symbolic execution frameworks and existing integer analysis techniques that symbolic differencing promises.

Loop trip count estimation computes or bounds the number of iterations a loop may perform. Sakellariou addresses the problem [92] by evaluating nested summations that involve symbolic upper and lower bounds, corresponding to the loop bounds, along with the stride of the linear induction variable. Summations are normalized to summations of polynomials and each normalized summation is annotated with a series of constraints on the bounds under which the normalization is valid, resulting in a set of trip count estimations.

Loop timing estimation is a generalization of loop trip count estimation that attempts to bound the execution time of a loop. Van Engelen and Gallivan propose [109] a method of bounding the sum of the time required for each loop iteration by the interpolating polynomial produced from Newton's forward formula for interpolation, which reduces summations with symbolic bounds to summations with only constant bounds, simplifying evaluation.

Within a system capable of symbolic differencing [46], trip count estimation for loops governed by linear induction variables follows directly from analysis of the loop termination condition. Loop timing estimations may be achieved by introducing an

integer variable with value zero prior to the loop and with increment operations on this variable for each operation in the loop that is to be counted [46]. In the absence of complicating factors such as conditional branches within the loop, this method returns precise loop timing estimates.

2.1.4 Higher-level static analysis

Object sensitivity [73] exploits a middle ground between context sensitivity and context insensitivity for object-oriented languages by being sensitive only to the implicit object parameter of a method call, instead of sensitive to all parameters in a subroutine invocation. Empirical evidence is presented illustrating that object sensitivity is significantly more precise than a context-insensitive analysis, but remains efficient for practical use.

Related field analysis [1, 39] attempts to model and prove intra- and inter-object field relationships. Related field analysis detects program invariants relating fields in two objects by verifying that the invariant is maintained by all subroutines that may access the fields in question, using Java's access control model, including `protected` and `private`, to limit the analysis only to subroutines that may affect the fields. This thesis describes (in Chap. 5) a technique with goals similar to the one that motivates related field analysis, but with one important difference: the relationships amongst fields that guide the analysis described in this thesis are *transient*, i.e., they can only be guaranteed to hold at particular program points, and are therefore not provable via related field analysis.

Role analysis [65] involves verification of the referencing relationships among a group of objects based on a specification of the "roles" each object will play. Roles capture important data structure properties that are not obvious from the class definitions, and present them via the role reference diagram. The role reference diagram is similar to the field relation graph described in Sec. 5.5 of this thesis. However, the former describes reference relationships whereas the latter represents numerical relationships, incorporating reference relationships via graph edge properties.

2.2 Static checking

As discussed in Sec. 1.1.3, static checking is the application of static analysis to the problem of finding bugs in program code. Static checkers verify that particular safety, security, or correctness properties hold within a program, and detect errors such as potential deadlocks in multithreaded code, use of undefined variables, non-portable type casting, and out-of-bounds array accesses. Some static checkers are extensible, allowing annotations within programs that introduce additional forms of checking and documentation, often via pre- and post-conditions [48]. We further characterize the two common approaches to static checking, language-based and specification-based checking.

2.2.1 Language-level checking

Language-level checkers detect bugs in the use of a particular programming language. The best-known static checker, `lint` [56], performs checking for various problems with programs using the C language. The field has grown considerably, encompassing many varied forms of analysis and checking to detect program bugs [24, 25, 28, 31, 32, 41, 58, 68, 99].

Static detection of security and safety problems in C programs generally focuses on strings and buffers, which are known to cause a large number of security vulnerabilities [25, 32, 66, 99, 116]. This form of checking often recognizes an extended form of the C language, for instance by including common library routines within the language to generate integer constraints [25, 116] or extending the language with additional type qualifiers [36, 99]. These uses of language extensions differ from our own approach to library-level analysis in that they arbitrarily select a set of library routines to elevate to language primitives within the static analysis, instead of providing extensible support for all libraries. Wilander and Kamkar provide a recent survey of publicly-available static security tools [118].

Array-bounds checking (and the elimination of run-time array bounds checks) is a well-studied problem [9, 14, 63, 70, 107] akin to the detection of buffer overruns. We are interested in the problem of determining whether an index falls within the bounds of a container object (such as an STL `vector` or deque) or determining

whether an iterator has been moved beyond the end of its sequence, which is similar to the array-bounds checking problem in many ways. The primary difference lies in the relative stability of array sizes and container sizes. While array sizes change only very rarely within a program and are often static, containers provided by software libraries, such as linked lists and binary trees, change size very often, complicating the process of loop analysis. These complications will be further discussed in Chap. 6.

2.2.2 Specification-based checking

A common implementation technique to construct extensible static analyses is a specification-based approach, wherein the user provides some formal specification of the intended behavior of an abstract data type or routine. The information can be used either to verify that an implementation conforms to its specification [6, 31, 68, 117] or to verify correct use of the data type or routine within user programs [16, 28, 31, 42]. This thesis focuses on the latter form of specification-based checking.

The Metal [28] language provides a unique view on checking against specifications. Specifications are user-defined state machines; transitions occur when patterns syntactically match the code. Each specification is applied on every execution path in a subroutine, to verify properties such as "If interrupts are disabled, they must be re-enabled before exiting the subroutine." The authors' approach does not conservatively check programs (i.e., it may not diagnose all errors), but due to its simplicity it can be applied to complex systems (e.g., the Linux and OpenBSD operating system kernels). Although the input language for checking is C++, the notion of syntactic pattern matching (only) for checking does not scale well with abstraction. Operator overloading, for instance, can produce a situation such that a syntactic expression `a = b` may not have semantics with even a passing resemblance to the semantics of integer or pointer assignment. Our work shares with Metal the notion that specifications need not be formal or even complete to effectively find problems in program code, along with the use of symbolic execution.

Ramalingam et al. describe a method of deriving program analyses [85], used in the Canvas project [16], that can efficiently verify component-client conformance,

which they apply to the Concurrent Modification Problem (CMP) in the Java Collections Framework (JCF) [17]. JCF contains iterators conceptually similar to those in the C++ STL, and the CMP refers to the use of an iterator that references a container that has been modified. The problem description is nearly identical to our own, although the details differ dramatically: within the JCF, iterators are invalidated by any operation on the container that does not occur through the iterator, thereby allowing the problem to be reduced to a versioning problem. The semantics of iterators in the C++ STL are more deeply tied to the underlying data structures, and iterators 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. The 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 a more expressive form for iterators than is needed for iterators in the Java Collections Framework.

Extended Static Checking (ESC) [24, 34, 30] uses incomplete programmer-written specifications of program code and unsound verification methods to find various common problems in programs. Specifications may introduce abstractions for use only in static checking, performing modifications to and checking of the abstract state, in a manner similar to the executable specifications presented in this thesis. ESC employs a theorem-proving method to detect program bugs, whereas this thesis discusses more traditional, symbolic-execution—based static analysis methods to detect misuse of software libraries in C++ programs.

The Alloy modeling language [51] expresses relationships amongst program objects via a relational first-order logic. Alloy has been employed for analysis of Java annotations [58] in a language similar to the Java Modeling Language [69], to the verification of object models [52]. Of particular interest for this thesis is the use of Alloy to address the concurrent modification problem in Java, using the versioning formulation introduced by Ramalingam et al. [85]. Alloy shares with Extended Static Checking the use of a theorem prover to find counterexamples that

violate assertions given about the program state. Due to limitations on the search for counterexamples, analyses based on Alloy are also unsound and may not detect some program errors.

Splint [105] (formerly LCLint [31]) allows the programmer to introduce new attributes [32] for variables and function parameters, similar to those implied by the automata used in Metal [28]. Program code may be annotated with checks verifying that particular attributes are present on a program variable and assertions when certain attributes are introduced. The introduction of semantic properties described in this thesis employs a similar methodology, although we require more expressive semantic properties; see Chap. 7.

2.3 Generic programming

Generic programming [8, 55, 82], also known as requirement-oriented programming [75], is a programming methodology that emphasizes the construction of reusable algorithms and data structures that are based on concepts [57, 78, 79], not on concrete data types. The inherent complexity of designing a generic software library lies in the appropriate decomposition of the important data types and abstractions into concepts, to achieve maximal reusability without sacrificing efficiency.

Generic programming achieved its first mainstream success with the introduction of the Standard Template Library (STL) [81, 106] into the C++ standard [5], and has continued to gain acceptance among C++ programmers [3, 8, 62]. Much of the appeal of using C++ for generic programming is due to template metaprogramming [22, 111], which allows one to exploit the template machinery of the C++ language to perform compile-time computation and optimization.

2.3.1 Concepts

Concepts, as previously mentioned, are composed of two sets: a set of requirements and a set of abstractions that meet those requirements. Different concepts often contain similar requirements: when the requirements of concept A are a subset of the requirements of concept B—and, therefore, the set of abstractions for the

concept A is a superset of the set of abstractions for concept B—we say that B *refines* A [57]. Given this partial ordering, related concepts are often described with lattices [120].

To date, concepts have no explicit representation within any mainstream programming language, requiring concept designers to resort to informal description models [5, 102, 119]. The Tecton concept description language [57, 78] expresses concepts and concept refinement within the formal framework of order-sorted algebras, which has been used to express the concepts of the C++ STL [79].

Some aspects of concept specification in C++ have been addressed via convention and clever use of C++ language facilities. Concepts are often represented via "tag" types, accessible via traits [83], where each tag type is a unique class and where C++ inheritance of tag types represents the refinement relation [5]. Concepts can be described more specifically via traits by providing additional associated types and constants via template arguments, as in the transformation of Tecton algebraic concepts into C++ [93]. The syntactic component of concept specifications, allowing a generic algorithm to verify that its parameters meet certain syntactic requirements and allowing one to verify the syntactic correctness of a generic algorithm with respect to its stated requirements, has been addressed via concept checking [101, 119].

2.3.2 Run-time checking of STL implementations

Several popular implementations of the STL include a "debug mode" for STL components [35, 38, 71], fashioned after Safe STL [49]. The debug mode for each of these libraries introduces additional run-time checking of algorithm preconditions, and therefore detects incorrect usage of the STL when it first occurs.

Of particular interest in run-time debug modes is the introduction of "safe" iterators, which exhibit the same behavior as normal iterators but maintain an explicit, bidirectional link with the containers they reference. This link is used to verify the validity of the iterators (e.g., the container signals its iterators when they become invalid) and to check operations such as dereferencing and movement.

While static checking of STL usage has many similarities with the run-time checking performed by these implementations, there are interesting differences. Static

checking in general provides multiple benefits over run-time checking, including the absence of run-time penalties associated with checking and the ability to analyze all program paths simultaneously, to detect bugs that occur only under very specific conditions. With this problem in particular, the run-time cost of checking is very high, as operations that require no computation without checking, such as iterator invalidation, become linear in the number of iterators in the presence of checking. Moreover, some properties that can be checked by a static analysis (such as "removedness" of values in sequences) cannot be provided with run-time checking without compiler support. Finally, run-time checking is implementation-specific and not widely available.

CHAPTER 3
Static checking for generic software libraries

Generic software libraries seek to provide the most general, most reusable, and most customizable solution to a particular problem without sacrificing efficiency or usability. The generality of generic software libraries, like other types of software, is achieved through the introduction of new (possibly domain-specific) abstractions, although the concepts employed by generic libraries are both more pervasive, allowing a greater degree of customization, and more abstract than in traditional software libraries.

Static checking for generic software libraries differs from static checking for user programs or traditional software libraries due to this greater use of abstractions. A static checker for generic libraries must cope with many abstractions, utilizing them to provide precise analysis and domain-specific diagnostics.

This chapter discusses the methodology applied to perform higher-level static checking for generic software libraries. Although this chapter and the examples throughout this thesis focus on the abstractions of the C++ Standard Template Library, the techniques discussed are in no way limited to the STL and are generally not even limited to the C++ language.

3.1 Overview of the Standard Template Library

The STL is a prime example of a generic software library. It is reusable, allowing the user to integrate user-defined data types with its components; it is customizable, permitting changes to algorithm semantics and object relationships; and it is extensible, enabling users to provide their own data structures that interoperate with STL algorithms, so that a user need not reimplement any of the algorithms in the STL for a new data structure. Additionally, any new data structure will also interoperate with new algorithms written by other users, so long as users conform to the concepts in the STL.

The major components of the STL are containers, iterators, and algorithms.

This section will describe these components and their interactions. Readers familiar with the STL may wish to skip to Sec. 3.2. For additional information regarding the STL, there are many books discussing the design and use of the STL [8, 81].

3.1.1 Containers

Containers in the STL are data structures that can store and retrieve values. Each container is implemented as a C++ class template that can be instantiated with any data type that meets certain minimum requirements and then stores values of that type. For instance, the container `vector<int>` stores values of built-in type `int`, whereas `vector<Person>` stores values of user-defined type `Person`.

There are several different kinds of containers within the STL, divided into "sequence" containers, which store elements in a linear fashion, and "associative" containers, which store elements in fast lookup/retrieval order. The STL contains three sequence containers: `vector`, a dynamically-resizeable array; `list`, a doubly-linked list; and `deque`, a double-ended queue. The `basic_string` class template used for strings in C++ [5, §21], also meets the SEQUENCE requirements [5, §23.1.1] but is not strictly part of the STL. The STL also contains four associative containers: `map`, a mapping from keys to values such that there exists zero or one values for each key; `multimap`, a mapping keys to values that permits zero or more values for each key; `set`, a set of (unique) values; and `multiset`, a multiset of values. The associative containers are sorted based on a (user-selectable) ordering predicate, and are typically implemented via a red-black tree.

Despite the wide variety of container types in the STL, their many similarities often make it possible to change from one container type to another without altering a large amount of code, although the underlying change in semantics must be carefully considered [72]. More importantly, however, all container types provide *iterators* allowing one to visit and modify the elements in a container without regard to the underlying data structure.

3.1.2 Iterators

Iterators in the STL are abstract data types that allow one to read, write, or modify the elements within a sequence of values without relying on knowledge

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

Figure 3.1: Implementation of the STL `find_if` algorithm

of the generation, consumption, or storage of those values. Syntactically, iterators are similar to C/C++ pointers, allowing at a bare minimum use of the `*` operator for dereferencing (to access the value the iterator refers to), the `++` operator for incrementing (to move to the next value in the sequence), and the comparison operators `!=` and `==` (to determine if the iterators are interchangeable). The loop in Fig. 3.1 illustrates the basic syntax used by iterators.

The state of a particular iterator determines what operations are well-defined on the iterator. The state will change throughout the course of the program's execution based on operations on the iterator, and based on operations on the underlying data structure (if any). The state of an iterator is generally described by one of the following:

- *Singular*: An iterator is singular when it does not refer to a valid position within a sequence. The only operations that are well-defined for a singular iterator are the copy operation (when the singular iterator is overwritten with a nonsingular value) and destruction.

- *Dereferenceable*: An iterator is dereferenceable when it references a valid element within a sequence, implying that the application of the dereference operator `*` is well-defined. A dereferenceable iterator is also *incrementable*,[1] meaning that one can apply the `++` operator to the iterator to advance it to the next position in the sequence. A dereferenceable iterator is not singular.

- *Past-the-end*: A past-the-end iterator is neither singular nor dereferenceable. Past-the-end iterators serve as markers for the end of a sequence. Iterators in a

[1] Other iterator concept taxonomies, such as the only used in the Silicon Graphics implementation of the STL [8, 98], do not necessarily require this equivalence.

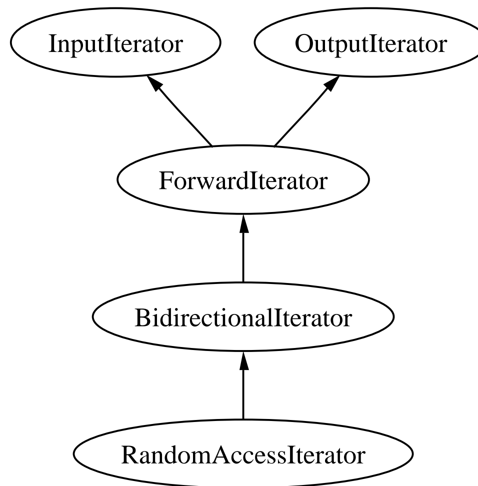


Figure 3.2: The lattice of STL iterator concepts

sequence that are incremented until they are no longer incrementable become past-the-end iterators.

Iterators are rarely useful by themselves, but are often found in valid range pairs. A valid range [`first`, `last`) is a pair of iterators such that some finite number (possibly zero) of applications of the increment operator `++` to `first` will result in `first == last`. Given a valid range [`first`, `last`), one may dereference all iterators `first`, `++first`, up to (but not including) `last`. Thus `last` may in fact be a past-the-end iterator. The algorithm in Fig. 3.1 iterates through all elements in the input valid range [`first`, `last`), searching for an element that satisfies the predicate.

Iterators in the STL model one of five iterator concepts illustrated by the concept lattice in Fig. 3.2. Each concept specifies, either directly or through refinement, how the iterator may move and how the values referenced by that iterator may be used. The iterator concepts [5, 8, 81] are:

- InputIterator: An input iterator is an iterator that can only move forward (via the `++` operator), passing through the sequence of values once to read them.

- OutputIterator: An output iterator is an iterator that can only move forward (via the `++` operator), writing to each of the values only once.

- ForwardIterator: A forward iterator is a refinement of both input and output iterators, so that the values accessed by forward iterators can be both read and written. In addition, forward iterators are *multi-pass* iterators, meaning that one can pass through the sequence of values more than once.

- BidirectionalIterator: A bidirectional iterator is a forward iterator that can also move backwards via the `--` operator.

- RandomAccessIterator: A random access iterator is a bidirectional iterator that can move an arbitrary distance in either direction in constant time.

Each STL container defines its own iterator types and two member functions, `begin` and `end`, which delimit the sequence of elements within that container. Given a container `c`, [`c.begin()`, `c.end()`) forms a valid range over all elements in the container. The concept that the iterator models depends on the STL container: `vector` and `deque` can support random access iterators, whereas the other STL containers support only bidirectional iterators.

By decoupling the way in which elements in a container are accessed from the data structure the container implements, iterators allow the creation of generic algorithms that can be instantiated for any kind of data structure supporting iteration, without loss of efficiency.

3.1.3 Algorithms

The STL contains more than 70 generic algorithms, including sorting, searching, sequence transformation, set, and heap operations. Each algorithm accepts one or more valid ranges whose iterators model the concept appropriate for that algorithm. The heapsort routine specified in Fig. 1.1, for instance, requires RandomAccessIterators because popping an element from a heap in logarithmic time requires constant-time access to any element in the heap. The searching algorithm in Fig. 3.1, however, requires only InputIterators, because it makes only one pass through the sequence. Thus the `find_if` algorithm can be applied on the elements of any STL container, and even for iterators (such as stream iterators [5, §24.5]) not related to containers.

```
08 vector<int>::iterator
09 insert_in_sorted_vector(vector<int>& values, int val)
10 {
11   vector<int>::iterator lb =
12     lower_bound(values.begin(), values.end(), val);
13   values.insert(lb, val);
14   return lb;
15 }
// ...
25 vector<int>::iterator pos = insert_in_sorted_vector(v, 42);
26 v.insert(pos, 41);
```

Figure 3.3: [`sorted_vec_insert.cpp`] **The iterator `lb` may be invalidated by the call to insert, but it only occurs when the size of the `vector` equals its capacity on entry.**

STL algorithms are often parameterized by *function objects*, i.e., objects that may be invoked via an overloaded function call operator and may have state. These function objects provide ordering relations (for sorting, set, and heap operations), predicates (for finding matches), and combinators (for transforming elements of one or more sequences). These function objects are an integral part of the semantics of STL algorithms.

3.2 Static checking for the Standard Template Library

Static checking for the Standard Template Library involves checking for proper use of the containers, iterators, and algorithms that are part of the STL. This thesis is concerned primarily with checking the following semantic conditions:

- *Iterator operations*: An STL checker should verify that iterator operations are safe. For instance, an iterator should not be decremented past the beginning of the sequence or incremented beyond the past-the-end location for that sequence. This checking should include verifying the domain of relational operations on iterators, such as `==` or `<`. For instance, two `istream_iterator` objects may not be compared if they do not reference the same underlying stream because the stream establishes the domain [5, §24.5.1].

- *Iterator invalidation*: Certain operations on containers or iterators *invalidate*

```
06 list<int> l1;
07 list<int> l2;
08
09 list<int>::iterator i = l1.end();
10 l2.insert(i, 17);
```

Figure 3.4: [`insert_other.cpp`] **The call to `insert` is ill-formed because the iterator `i` refers to the wrong container.**

an iterator, meaning that the iterator may become singular due to that operation. An example of such an operation is erasure of the element referred to by an iterator, which invalidates that iterator. Invalidation rules can be subtle: the function in Fig. 3.3 will be incorrect only when the size and capacity of the given `vector` are equal, because insertion into a `vector` may cause iterators referencing it to be invalidated. The problem manifests itself as an invalid operation on a singular iterator. Our high-level static checker, STLlint, can detect this error and will produce the following diagnostic:

```
"sorted_vec_insert.cpp", line 14, warning: attempt to copy a
      singular iterator

    return lb;
```

- *Container operations*: Container operations often have preconditions based on the state of the container. For instance, the `front` and `back` functions for Sequence containers [5, §23.1.1], which return references to the first and last elements, respectively, require that the container be nonempty. Additionally, like built-in arrays, some containers have subscripting operations that require bounds checking.

- *Iterator/container interaction*: Iterators are used in many container operators to refer to the container's elements. For instance, insertion before or deletion of a particular element requires that one supply an iterator referencing that element to the container's `insert` or `erase` operation. In this case the iterator must actually reference that container, a property that should be verified to

```
09  vector<double> v;
    // ...fill v...
14  // Remove zeroes
15  remove(v.begin(), v.end(), 0.0);
16  // Calculate inverses
17  vector<double> inv_v;
18  transform(v.begin(), v.end(), back_inserter(inv_v),
19            bind1st(divides<float>(), 1.0));
```

Figure 3.5: [`erase_remove.cpp`] **The invocation of the `remove` algorithm reshuffles elements, but does not actually remove them, so the `vector` `v` contains some elements with undefined value.**

expose bugs. STLlint is again capable of detecting these errors, producing the following diagnostic for the example in Fig. 3.4:

```
"insert_other.cpp", line 10, warning: attempt to insert into
      container with an iterator from a different container

    l2.insert(i, 17);
```

- *Algorithm properties*: STL algorithms have many requirements, specified both explicitly and implicitly, as illustrated in Fig. 1.1. They often introduce semantic properties on (sub)sequences denoted by iterators, such as sorting the sequence, arranging it in a heap, or removing elements by giving them undefined values. Fig. 3.5 illustrates one instance where tracking the subsequence of elements in a container that have been removed by the `remove` (or `unique`) STL algorithm is required to detect potentially unsafe behavior:

```
"erase_remove.cpp", lines 18-19, warning: some elements in the
       iterator sequence may have been removed but have not
       been erased

      transform(v.begin(), v.end(), back_inserter(inv_v),
                bind1st(divides<float>(), 1.0));
```

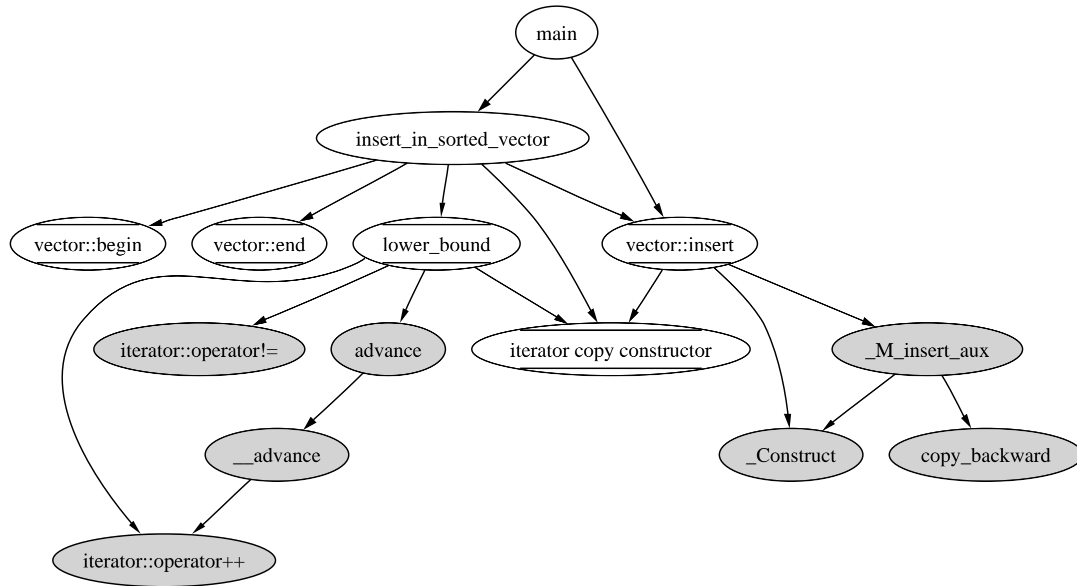


Figure 3.6: Partial call graph for the program snippet in Fig. 3.3. User code is represented by empty nodes, library implementation code is represented by shaded nodes, and the remaining nodes represent interfaces between library and user code.

3.3 High-level static checking methodology

The approach to high-level static checking described in this thesis splits programs into two segments, user code and library code, separated by the library interface. We transform the user code into a new imperative analysis language, Semple, whereas the library code is replaced completely with a lightweight specification (also written in Semple) of the interface-level semantics of the library. Fig. 3.6 illustrates the separation of a program's call graph into these two segments, highlighting the interface layer between user code and library implementation code. All code represented by shaded nodes will be replaced with lightweight specification code implementing the interface-level semantics.

Once the library has been replaced with its specification, we have a representation of the complete user program relative to the library semantics. Given this representation, we employ a whole-program analysis based on symbolic execution, paying particular attention to program loops, to verify the correctness of the user program with respect to the library semantics.

3.3.1 Standard Template Library specifications

Although this thesis explores high-level static checking in general, the application of primary interest is the STLlint static checker for the STL. This section discusses the specification models of containers and iterators that will be referenced throughout the discussion.

The implementations of STL containers vary widely, but even so the interface-level semantics are very similar. For instance the sequence containers, `vector`, `list`, and `deque`, have many common operations that differ only in algorithmic complexity and iterator invalidation semantics. Even the associative containers have very similar semantics to the sequence containers, with iterator invalidation rules that are very similar to those of `list`. The most important property for containers in the STL is in fact a trivial one: the number of elements in the container, called the container's *size*. Insertion increases the size of the container, while erasure decreases it; routines such as `front` and `back` require that the size is nonzero, whereas subscripting operators for `vector` and `deque` require bounds checking based on the size. In addition to *size*, we also provide for each container a positive, nonzero integer *version* that is initialized to one and is incremented whenever all iterators in the container are invalidated simultaneously.

As with containers, the interface-level semantics of iterators can be implemented in many ways. Iterators can move through the elements of a sequence by stepping forward, stepping backward, or even skipping an arbitrary number of steps in either direction. Often, it is important to determine whether two iterators form a valid range and, if so, how many times the first iterator would have to be incremented to reach the last (i.e., the *distance* between the iterators). To capture the notion of iterator motion through a sequence, we specify two further attributes for iterators: a reference (or pointer) to the *sequence* (or container) the iterator references and an integral *position* in the container: position zero corresponds to the first element in the sequence whereas a position equal to the size of the container corresponds to the past-the-end iterator for that container. With this representation, two iterators `first` and `last` form a valid range if they refer to the same sequence and if `first.`*position* $\leq$ `last.`*position*. To permit checking of singular iterators

and implement iterator invalidation, all iterators contain a positive integer *version* that denotes the container version for which the iterator is valid. An iterator is nonsingular when its version is equal to the version of the container it references.

Unlike component libraries such as the Java Collections Framework [17], not all container modifications affect the container version. For containers that use contiguous storage, such as `deque` and `vector`, operations that may force reallocation of storage (such as insertion) may increment the version number. However, containers such as `list`, (`multi`)`map`, and (`multi`)`set`, only increment the version number when the container itself is destroyed. Many more container operations invalidate only a few iterators (by setting the iterator's *version* to zero), based on the element positions involved in the operation.

Specifications of STL algorithms are generally achieved via an event model discussed in Sec. 7.2. For algorithms with very simple implementations (as in the `find_if` algorithm of Fig. 3.1) it is sufficiently efficient and precise to analyze the algorithm's implementation. With more complicated algorithms, however, we provide a complete specification that mimics the externally-visible postconditions while avoiding the cost (in both run time and precision) of analyzing the entire implementation. This will be discussed further in Chap. 7.

3.3.2 Symbolic execution

Symbolic execution [61] is a program analysis method that simulates the execution of program code on an *abstract* program state, as discussed in Sec. 2.1.1. Within the abstract program state, the values of each program variable and each value allocated on the heap are represented symbolically, such that the symbolic value conservatively approximates all real values that variable (or location) may contain.

Consider the program in Fig. 3.7, which computes a grade based on two test scores. If this program were to be executed normally, the variables `test_grade` and `retest_grade` would be supplied with integral values typed by the user. However, with symbolic execution we must assume that the values of these variables can be any integer (because they are part of the program's input), and we therefore

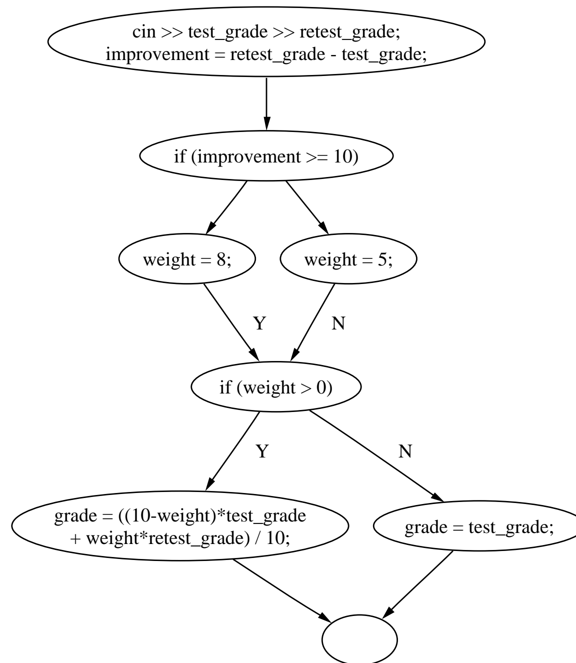


Figure 3.7: A simple program to compute an overall grade

assign the symbolic values m and n as the values of `test_grade` and `retest_grade`, respectively. At the next line, we compute the *symbolic* value of the expression `retest_grade - test_grade` to be $n-m$, and associate this value with the variable `improvement`. Note that, at this point, we could substitute actual values for m and n to reproduce a particular execution of the program.

Symbolic execution differs from normal execution particularly at branches in the control flow. While a normal execution of the program would select one path out of the `if` statement in Fig. 3.7, a symbolic execution may not have enough information to statically determine the path to take. In the original formulation of symbolic execution [61], the testing tool would prompt the user to select a branch to take or would consider all branches. In this thesis we will instead adopt the control-flow merging behavior of abstract interpretation [20], where the analysis executes both paths independently. The results attained from both paths are then merged into a conservative approximation of both paths. In the example, the value for the variable `weight` will be 8 on the `then` branch and 5 on the `else` branch, which will be merged into some symbolic value that approximates the values 5 and 8.

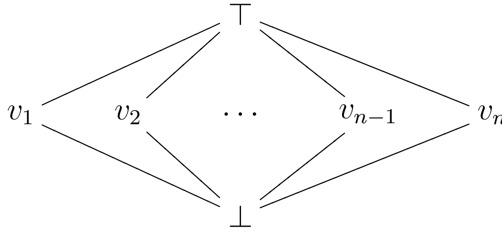


Figure 3.8: Trivial join lattice used to construct a symbolic domain from a concrete domain that consists of values v_1, v_2, ..., v_n

The principal components of a symbolic execution framework are the modules that represent and manipulate values in the abstract program state for a particular data type. To perform symbolic execution of the simple program in Fig. 3.7 requires an integer analysis module capable of addition, subtraction, multiplication, division, comparison, and merging of symbolic integer values. Languages with other types, such as pointers, booleans, and floating-point numbers, require additional modules. Symbolic execution naturally fuses these modules together, evaluating the program via one consistent analysis, thereby eliminating the phase ordering problem generally associated with independent analyses. For instance, symbolic execution naturally performs dead code elimination, because it will find no path to unreachable code. In illustration, the second `if` condition in Fig. 3.7 will evaluate true for any nontrivial integer analysis, and the symbolic execution will execute only the `then` branch, ignoring the `else` branch. A more traditional analysis, which does not fuse multiple analyses together, would require an integer analysis phase to determine that `weight > 0` always holds, followed by a dead code elimination pass, after which other optimizations and analyses may further improve the precision of the information.

In this thesis, the most important analysis module for a data type within the symbolic execution framework is the integer analysis module. This module is responsible for the analysis of iterator positions and is discussed in Chap. 5. The analysis modules for other data types in the Semple language are implemented as with abstraction interpretation using the trivial join lattice illustrated in Fig. 3.8, where $v_1, v_2, \ldots, v_n$ are the values within the type's domain.

3.3.3 Program loops

The original formulation of symbolic execution [61] supports program loops by querying the user at each conditional branch, requiring the user to determine how many times the loop should execute within a particular test. Symbolic execution can therefore analyze program loops automatically only when it can determine that the number of iterations is statically bounded, i.e., when there is a finite number of paths in the program. Abstract interpretation, on the other hand, iterates to a fixed point using *widening* when the symbolic domain is a lattice of infinite height and *narrowing* to recover information from conditional branches [15, 20].

Neither the symbolic execution solution (which fails on many program loops) nor the abstract interpretation solution (which may produce very imprecise bounds) is adequate for analysis of iterator loops such as that of Fig. 3.9 (extracted from an introductory C++ text [62], where it is described in some detail). In this loop, the iterator `iter` moves forward either zero or one steps in each iteration, whereas the size of the container either remains constant or is decreased by one by the `erase` operation. However, because the loop termination condition involves function calls, indirection via pointers, and copies of induction variables, existing approaches to narrowing cannot adequately restrict the positions of iterators in the loop, and an abstract-interpretation–based analysis produces imprecise information.

This thesis details an approach to loop analysis that combines symbolic execution with traditional induction variable recognition to simulate the effect of an arbitrary number of loop iterations in a single pass. The effect is neither abstract interpretation nor symbolic execution, although we adopt the latter term because it most specifically describes the approach. However, the resulting analysis is more precise than either abstract interpretation or symbolic execution, and is able to correctly verify the safety of the example in Fig. 3.9 with respect to the STL specifications.

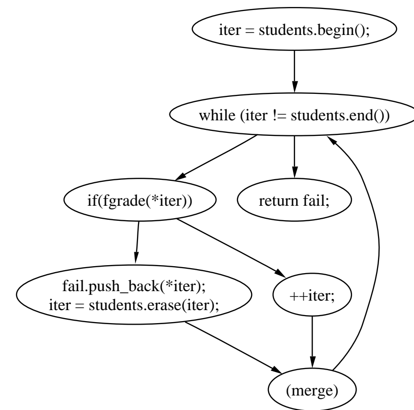
3.3.4 Recursion

Interprocedural symbolic execution is similar to the call string approach to interprocedural data flow analysis [100] in that it explicitly preserves the call context

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

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

(a) Extract failing students



(b) Control flow graph

Figure 3.9: Verification of a subroutine that erases students with a failing grade from a vector requires precise loop analysis.

for every subroutine invocation. Symbolic execution differs, however, in that it does not limit the length of the call string by a constant. For nonrecursive programs, this implies that every node in the invocation graph [27], which represents different calling contexts by different graph nodes, will be analyzed separately.

For programs involving recursive subroutines, the length of the call string is unbounded, requiring particular care to ensure termination. We formulate our recursive analysis in the vein of an abstract interpretation on an interprocedural control-flow graph. In an interprocedural control-flow graph, such as that of Fig. 3.10, subroutine calls are represented by call/return node pairs connected by a dotted edge. Dashed control-flow edges indicate interprocedural control flow, both from a call node to the entry of the function it invokes and from the exit node of a function back to the (corresponding) return node.

Recursion introduces cycles into our interprocedural control-flow graph at call/return edges. We select these interprocedural edges as *widening edges* [15, 20], meaning that traversal of the edge by the symbolic execution implies that the state at the target be widened by the state at the source. The widening operation is an overly conservative join operation that guarantees a finite number of iterations

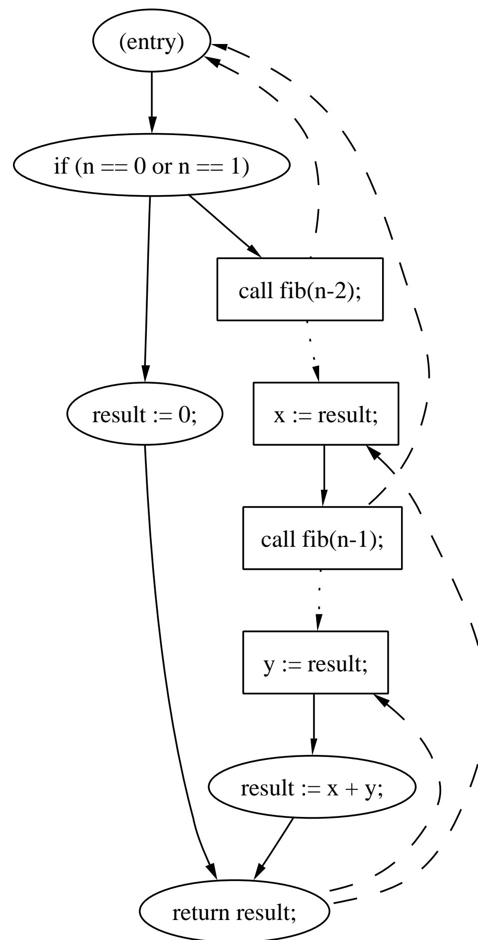


Figure 3.10: Interprocedural control-flow graph for the recursive formulation of the Fibonacci function.

before one reaches a fixed point, even when the join operation for the underlying domain (such as a lattice of infinite height) would not.

Within an abstract interpretation, where the state at each control-flow node is explicitly represented, the widening operation suffices to perform analysis of recursive functions on an interprocedural control-flow graph. Symbolic execution, on the other hand, represents only the active program states, requiring one to consider the potential for interprocedural widening edges in advance. To this end, on entry to a particular subroutine symbolic execution divides the program state into three distinct parts:

- *Working*: The working state is the program state that is to be continually

updated as the program executes. For a provably nonrecursive function, the working state is the only state that is necessary.

- *Input*: The input state is a snapshot of the program state on entry to any invocation of the (recursive) subroutine under analysis. It is initially equivalent to the working state, but will be widened on any recursive interprocedural call edge to this function's entry node.

- *Output*: The output state is a summary of the effects of this subroutine. It is initially empty, but will be widened whenever the symbolic execution processes the exit node for the subroutine. Additionally, the output state overrides values in the working set when the symbolic execution traverses edges from call nodes to the corresponding return nodes (the dotted edges in Fig. 3.10).

The use of the three distinct program states, *working*, *input*, and *output*, in conjunction with fixed-point iteration on the body of a subroutine permits interprocedural control-flow analysis of recursive subroutines. This formulation is equivalent to an abstract interpretation, but does not require representation of all program states simultaneously nor explicit representation of the interprocedural control-flow graph.

CHAPTER 4
The Semple language

Semple is a new internal representation language suitable for the representation of object-oriented programs and the specification of software libraries. The Semple language is sufficient to express the majority of programs implemented in a type-safe subset of C++, but provides additional primitives necessary for first-order, extensible specifications.

4.1 Motivation and goals

The design of the Semple language is driven by three primary goals. The first is the desire to accurately represent the semantics of C++ programs, while retaining the essential structure of the control flow and data types in the programs. The second goal is to provide support for specifications of algorithms and data structures, including extensible data structure specifications and first-order logic. Finally, the Semple language is an internal representation language, requiring it to be suitable for various forms of static analysis.

Semple provides a type system including basic integer, pointer, and boolean types along with user-defined structured data types (classes). The use of classes permits translation of object-oriented programs into Semple without eliminating high-level information about object relationships. Each Semple variable has a single, static type, but may also have additional, dynamic types ("classifications") that can store additional data and extend data structure specifications. Multiple dynamic classification permits function specifications to arbitrarily extend the specifications of other types, decoupling semantic checking from implementing specifications.

Semple is a simple, minimal language. While programming languages meant for programmers generally offer many ways to perform a given task, Semple presents very few options. For instance, Semple contains only a single looping construct (`while`), a single conditional branching construct (`if-then-else`), and permits side effects only in very restricted circumstances. Moreover, Semple is a block-structured

internal representation, such that each block is either adjacent to another block (executing directly before or after it) or nested within another block. Semple does not provide the ability to create arbitrary control flow, e.g., via a `goto` statement, although this limitation could be eliminated via a restructuring pass [29]. Thus all control-flow graphs derived from Semple programs are reducible.

As a specification language, Semple provides two statements for verifying that a particular condition holds: `assert`, which verifies that the given expression is never false, and `assume`, which assumes that the given expression is true. Semple also provides a looping mechanism, `foreach`, that iterates over all objects of a particular class type within the program state, allowing specifications to employ first-order logic.

Overall, the Semple language strikes a balance between simplicity and expressiveness, for representing both object-oriented user programs and software library specifications. Semple is the representation language employed by the STLlint static checker and will be used throughout this thesis for exposition.

4.2 Syntax and semantics

This section describes the syntax and semantics of the Semple language.

4.2.1 Lexical conventions

Identifiers, represented by the nonterminal IDENTIFIER, consist of the longest string of consecutive characters starting with a lowercase or uppercase letter (`a`–`z` or `A`–`Z`) or an underscore (‘_’) followed by any combination of lowercase or uppercase characters, underscores, and digits (`0`–`9`) that is not a reserved word. All keywords used in the grammar are reserved words.

Strings, represented by the nonterminal STRING, are an arbitrary set of non-newline characters delimited by double quotes (`"`). A double quote may occur within a string if it is escaped with a backslash (i.e., `\"`). A newline may occur within a string via the escape sequence `\n` and a backslash may occur via the escape sequence `\\`.

Whitespace in Semple programs separates tokens but is otherwise insignificant.

Comments, denoted by a `//` in the code and continuing to the next newline, are not significant.

4.2.2 Declarations

Declarations in Semple are classes, subroutines, and variables.

⟨*decl*⟩ ::= ⟨*class*⟩ | ⟨*subroutine*⟩ | ⟨*declaration*⟩

4.2.2.1 Classes

Classes in Semple are roughly equivalent to Pascal records or C structures. Classes contain a series of named data members (fields) of any defined type. Classes may not be directly recursive, but one may construct pointers to the class within the class definition.

⟨*class*⟩ ::= `class` IDENTIFIER '{' ⟨*attribute-list*⟩ '}'
⟨*attribute-list*⟩ ::= ⟨*attribute*⟩ ⟨*attribute-list*⟩ | ϵ
⟨*attribute*⟩ ::= IDENTIFIER '`:`' ⟨*type*⟩ '`;`'

4.2.2.2 Variables

Variable declarations may occur at global scope (where they introduce global variables) or as statements.

⟨*declaration*⟩ ::= IDENTIFIER '`:`' ⟨*type*⟩ '`;`'

4.2.2.3 Subroutines

Subroutines in Semple may be either functions, which return a value, or procedures, which do not.

⟨*subroutine*⟩ ::= `procedure` IDENTIFIER '(' ⟨*parameter-list*⟩ '`)`' ⟨*block*⟩
⟨*subroutine*⟩ ::=
    `function` IDENTIFIER '(' ⟨*parameter-list*⟩ '`)`' '`:`' ⟨*type*⟩ ⟨*block*⟩
⟨*parameter-list*⟩ ::= ⟨*nonempty-parameter-list*⟩ | ϵ
⟨*nonempty-parameter-list*⟩ ::=
    ⟨*parameter*⟩ | ⟨*parameter*⟩ '`,`' ⟨*nonempty-parameter-list*⟩
⟨*parameter*⟩ ::= IDENTIFIER '`:`' ⟨*type*⟩

4.2.3 Types

Semple has a limited number of basic types, including integers (either signed or unsigned), booleans, and pointers. The integer types differ from native integer types in C++ [5], both because Semple integer types have arbitrary length and because arithmetic on the Semple `unsigned` type corresponds to arithmetic on natural numbers, and saturates at zero by silently replacing any negative result with zero. Any name declared as a class may also be used as a type.

⟨*type*⟩ ::= `integer` | `unsigned` | `bool`
⟨*type*⟩ ::= '`^`' ⟨*type*⟩
⟨*type*⟩ ::= IDENTIFIER

4.2.4 Expressions

The `unknown` expression is an untyped expression that indicates an unknown value. The type of the unknown expression is inferred by its context.

⟨*term*⟩ ::= `unknown`

4.2.4.1 Boolean expressions

There are two boolean literals, `true` and `false`.

⟨*term*⟩ ::= `true` | `false`

Logical conjunctions and disjunctions may only be applied to boolean expressions, and are computed using short-circuit evaluation.

⟨*expression*⟩ ::= ⟨*expression*⟩ `or` ⟨*and-expr*⟩
⟨*and-expr*⟩ ::= ⟨*and-expr*⟩ `and` ⟨*compare-expr*⟩

Pointers, integers, and boolean values may be compared for equality or inequality. The left- and right-hand expression types must be equivalent, where the type of the `nil` pointer constant is equivalent to any pointer type and the `unsigned` and `integer` types are considered equivalent.

⟨*compare-expr*⟩ ::= ⟨*add-expr*⟩ `==` ⟨*add-expr*⟩

⟨*compare-expr*⟩ ::= ⟨*add-expr*⟩ `!=` ⟨*add-expr*⟩

Integer values (either signed or unsigned) may be ordered via the comparison operators.

⟨*compare-expr*⟩ ::= ⟨*add-expr*⟩ ‘`<`’ ⟨*add-expr*⟩
⟨*compare-expr*⟩ ::= ⟨*add-expr*⟩ `<=` ⟨*add-expr*⟩
⟨*compare-expr*⟩ ::= ⟨*add-expr*⟩ ‘`>`’ ⟨*add-expr*⟩
⟨*compare-expr*⟩ ::= ⟨*add-expr*⟩ `>=` ⟨*add-expr*⟩

The `is-a` operation provides the ability to determine if the given lvalue has a dynamic classification of the given type.

⟨*compare-expr*⟩ ::= ⟨*lvalue*⟩ `is-a` ⟨*type*⟩

Logical negation operates on values of type `bool` and returns a value of type `bool`.

⟨*unary-expr*⟩ ::= ‘`!`’ ⟨*unary-expr*⟩

4.2.4.2 Integer expressions

Integer literals consist of an optional ‘`+`’ or ‘`-`’ followed by a nonempty series of digits.

⟨*term*⟩ ::= INTEGER_LITERAL

Integer values (signed and unsigned) may be used in addition, subtraction, multiplication, division, and modulus operations. If either expression has type `integer`, the result is of type `integer`, otherwise the result is of type `unsigned` and arithmetic saturates at zero. If either operand to the modulus operator is negative, the result of the expression is undefined as in C++ [5, §5.6].

⟨*add-expr*⟩ ::= ⟨*add-expr*⟩ ‘`+`’ ⟨*mul-expr*⟩
⟨*add-expr*⟩ ::= ⟨*add-expr*⟩ ‘`-`’ ⟨*mul-expr*⟩
⟨*mul-expr*⟩ ::= ⟨*mul-expr*⟩ ‘`*`’ ⟨*unary-expr*⟩
⟨*mul-expr*⟩ ::= ⟨*mul-expr*⟩ ‘`/`’ ⟨*unary-expr*⟩

⟨*mul-expr*⟩ ::= ⟨*mul-expr*⟩ '`%`' ⟨*unary-expr*⟩

The integer negation operation always returns a value of type `integer`.

⟨*unary-expr*⟩ ::= '`-`' ⟨*unary-expr*⟩

4.2.4.3 Pointer expressions

The only pointer literal is the null pointer, named `nil`. Its type is a pointer to an unspecified, internal type reserved for the `nil` pointer itself. All `nil` pointers have the same type, and no other type may include this type.

⟨*term*⟩ ::= `nil`

Any pointer whose type is not the type of `nil` and whose value is not equal to the `nil` pointer may be dereferenced, yielding an lvalue. Any program that dereferences a pointer that is equal to the `nil` pointer is ill-formed.

⟨*lvalue*⟩ ::= '`@`' ⟨*expression*⟩

The address of any lvalue (of type `T`) may be taken, yielding a non-`nil` pointer of type `^T` such that dereferencing the pointer yields the original lvalue.

⟨*unary-expr*⟩ ::= '`&`' ⟨*lvalue*⟩

4.2.4.4 Lvalue expressions

A reference to any variable is an lvalue of the type of that variable.

⟨*lvalue*⟩ ::= IDENTIFIER

The arrow operator (`->`) accesses a named field within an lvalue of class type.

⟨*lvalue*⟩ ::= ⟨*lvalue*⟩ `->` IDENTIFIER

An lvalue of a particular type may be transformed to an lvalue of one of its dynamic classifications via the `as-a` keyword. An lvalue expression `x as-a T` is well-formed when the expression `x is-a T` evaluates `true`.

⟨*lvalue*⟩ ::= ⟨*lvalue*⟩ `as-a` ⟨*type*⟩

4.2.4.5 Miscellaneous grammar productions

These productions, while necessary to complete the expression grammar, are of little semantic interest.

⟨*expression*⟩ ::= ⟨*and-expr*⟩

⟨*and-expr*⟩ ::= ⟨*compare-expr*⟩

⟨*compare-expr*⟩ ::= ⟨*add-expr*⟩

⟨*add-expr*⟩ ::= ⟨*mul-expr*⟩

⟨*mul-expr*⟩ ::= ⟨*unary-expr*⟩

⟨*unary-expr*⟩ ::= ⟨*term*⟩

⟨*term*⟩ ::= ‘`(`’ ⟨*expression*⟩ ‘`)`’

⟨*unary-expr*⟩ ::= ⟨*lvalue*⟩

⟨*lvalue*⟩ ::= ‘`(`’ ⟨*lvalue*⟩ ‘`)`’

4.2.5 Statements

Blocks consist of a (possibly empty) sequence of statements.

⟨*statement*⟩ ::= ⟨*block*⟩

⟨*block*⟩ ::= ‘`{`’ ⟨*statement-list*⟩ ‘`}`’

⟨*statement-list*⟩ ::= ⟨*statement*⟩ ⟨*statement-list*⟩ | ϵ

The null statement does nothing.

⟨*statement*⟩ ::= ‘`;`’

Declaration statements introduce local variables, valid within the block in which they are introduced.

⟨*statement*⟩ ::= ⟨*declaration*⟩

Assignments overwrite the value on the left-hand side with the value of the expression on the right-hand side. The types of the left-hand side must be equivalent, by the same definition of equivalence used in the equality operators.

⟨*statement*⟩ ::= ⟨*lvalue*⟩ `:=` ⟨*expression*⟩ ‘`;`’

Call statements execute a subroutine by copying the value of the argument expressions into the formal parameters of the target subroutine. The assignment forms of call expressions are valid only when the target is a function with a return type equivalent to the lvalue on the left-hand side. A function with a return type may be invoked without assigning the result; in this case, the return value is ignored. Recursive subroutine calls are permitted.

⟨*statement*⟩ ::= ⟨*lvalue*⟩ `:=` IDENTIFIER ‘`(`’ ⟨*expr-list*⟩ ‘`)`’ ‘`;`’
⟨*statement*⟩ ::= IDENTIFIER ‘`(`’ ⟨*expr-list*⟩ ‘`)`’ ‘`;`’
⟨*expr-list*⟩ ::= ⟨*nonempty-expr-list*⟩ | ϵ
⟨*nonempty-expr-list*⟩ ::= ⟨*expression*⟩ ‘`,`’ ⟨*nonempty-expr-list*⟩ | ⟨*expression*⟩

Return statements immediately exit the current function, returning control to the caller. The return statement for a procedure must not have a value, whereas the return statement for a function must have an expression whose type is equivalent to the return type of the function. Functions must have a return statement on all paths.

⟨*statement*⟩ ::= `return` ‘`;`’
⟨*statement*⟩ ::= `return` ⟨*expression*⟩ ‘`;`’

Assertions and assumptions verify that particular conditions hold true. Assertions contain a text string that will be emitted as an error message should the assertion not hold.

⟨*statement*⟩ ::= `assert` ‘`(`’ ⟨*expression*⟩ ‘`,`’ STRING ‘`)`’ ‘`;`’
⟨*statement*⟩ ::= `assume` ⟨*expression*⟩ ‘`;`’

The `if-then-else` control statement evaluates its boolean condition, and executes either the `then` block (if the condition was `true`) or the `else` block (if the condition was `false`). The `else` block is optional.

⟨*statement*⟩ ::= `if` ⟨*expression*⟩ `then` ⟨*block*⟩
⟨*statement*⟩ ::= `if` ⟨*expression*⟩ `then` ⟨*block*⟩ `else` ⟨*block*⟩

The looping construct in Semple executes the loop body and then evaluates the boolean loop condition. When `true`, the process repeats until the loop condition evaluates `false`.

⟨*statement*⟩ ::= `do` ⟨*block*⟩ `while` ⟨*expression*⟩ '`;`'

The `new` and `delete` statements allocate and deallocate, respectively, memory from the heap. The `new` statement allocates memory from the heap for an object of the given type `T`, and returns a non-`nil` pointer of type `^T` that references the newly-allocated memory. The argument to `delete` must be a non-`nil` pointer that refers to memory allocated with `new` but not previously deallocated with `delete`.

⟨*statement*⟩ ::= `new` ⟨*type*⟩ '`;`'
⟨*statement*⟩ ::= `delete` ⟨*expression*⟩ '`;`'

The `classify` and `declassify` statements introduce and remove dynamic classifications for a particular object. When a `classify` statement is executed, a new classification of the given type is introduced into the set of classifications for the object referenced by the lvalue; if a classification of that type already exists, the statement has no effect. Conversely, when a `declassify` statement is executed, the classification of the given type for the object referenced by the lvalue is removed; if no classification of that type exists, the statement has no effect.

⟨*statement*⟩ ::= `classify` ⟨*lvalue*⟩ `as-a` ⟨*type*⟩ '`;`'
⟨*statement*⟩ ::= `declassify` ⟨*lvalue*⟩ `as-a` ⟨*type*⟩ '`;`'

The `foreach` loop iterates over the abstract program state stored by a static analysis, considering each object of the given type. The given identifier is introduced as a variable of the given type, but has no storage associated with it. Instead, the variable names each of the abstract locations with the given type, binding that location's address and executing the body of the loop. This statement permits first-order logic within Semple specifications, e.g., "for all values `x` of type `iterator`, increment `x.`*position*."

⟨*statement*⟩ ::= `foreach` IDENTIFIER '`:`' ⟨*type*⟩ ⟨*block*⟩

```
class vector {                                    class iterator {
  size : unsigned;                                  sequence : ^vector;
  guaranteed_capacity : unsigned;                   position : integer;
  version : unsigned;                               version : unsigned;
}                                                 }
```

Figure 4.1: Semple specifications of a `vector` and its `iterator` type

4.2.6 Translation unit

A translation unit contains a complete set of a Semple declarations, often stored in a file on disk.

$\langle$*translation-unit*$\rangle$ ::= $\langle$*decl-list*$\rangle$

$\langle$*decl-list*$\rangle$ ::= $\langle$*decl*$\rangle$ $\langle$*decl-list*$\rangle$ | ϵ

4.3 Specification example

Fig. 4.1 illustrates a specification of the STL `vector<int>` data type and its associated `iterator` data type. The *size* and *version* fields of the `vector` specification have already been discussed; the *guaranteed_capacity* field is a `vector`-specific quantity that dictates when reallocations occur, which affects iterator invalidations [42].

The specification of the `insert` method of an STL `vector` [5, §23.4.2.3] is given in Fig. 4.2. Fig. 4.3 illustrates the corresponding Semple `insert` function specification. Semple assertions at lines 6–11 are employed to verify that the `pos` iterator passed to the `insert` function is nonsingular and refers to this particular vector. We use two auxiliary tag classes to maintain information about sorted sequences. The `sorted` tag applies to sequences whose elements are sorted and the `sorted_insert_position` tag applies to iterators that are the result of binary search operations, such as `lower_bound`. The example specification describes the semantics of insertion with respect to sorted sequences: if the iterator position at which the insertion occurs is the result of a `lower_bound` operation, it preserves the sorted nature of the container; otherwise, the sequence is no longer sorted and the `sorted` classification is removed by line 14. Chap. 7 further describes the role of multiple dynamic classification in checking higher-level properties like sortedness.

```
iterator insert(iterator position, const T& x);

void insert(iterator position, size_type n, const T& x);

template <class InputIterator>
  void insert(iterator position,
              InputIterator first, InputIterator last);
```

1. **Notes:** Causes reallocation if the new size is greater than the old capacity. If no reallocation happens, all the iterators and references before the insertion point remain valid. If an exception is thrown other than by the copy constructor or assignment operator of `T` there are no effects.

2. **Complexity**: If `first` and `last` are forward iterators, bidirectional iterators, or random access iterators, the complexity is linear in the number of elements in the range [`first`, `last`) plus the distance to the end of the vector. If they are input iterators, the complexity is proportional to the number of elements in the range [`first`, `last`) times the distance to the end of the vector.

Figure 4.2: Specification of the `vector` `insert` member function excerpted from the C++ standard [5, §23.4.2.3]

The `insert` specification in Fig. 4.3 performs iterator invalidations as necessary. When the *size* and *guaranteed_capacity* of a `vector` are equal on insertion, all iterators are invalidated at line 20 by updating the version number of the `vector` and the capacity of the vector is (conservatively) increased to accomodate the new element. On the other hand, when the capacity of the vector exceeds the new *size*, the `foreach` loop at lines 23–26 selectively invalidates all vector iterators that reference the element at or after the insertion position. Chap. 5 discusses the static analysis methods capable of evaluating the relationships between iterator positions, container sizes, and the `vector` capacity. Chap. 6 discusses analysis of these specifications within program loops.

```
01 class sorted {}                 // The sequence is sorted
02 class sorted_insert_position {} // The result of lower_bound, etc.
03
04 iterator insert(this : ^vector, pos : iterator, x : integer)
05 {
06   assert(pos->sequence != nil
07          and pos->version == (@pos->sequence)->version,
08          "attempt to insert into container with a singular iterator");
09   assert(pos->sequence == this,
10          "attempt to insert into container with iterator from a
11           different container");
12
13   if (@this is-a sorted) then {
14     if not (pos is-a sorted_insert_position) then {
15       declassify @this as-a sorted;
16     }
17   }
18
19   if (size + 1 > guaranteed_capacity) then {
20     version := version + 1;
21     guaranteed_capacity := size + 1;
22   } else {
23     foreach i : iterator {
24       if (i->sequence == this and i->position >= pos.position) then {
25         i->version := 0;
26       }
27     }
28   }
29   size := size + 1;
30
31   pos->version := version;
32   return pos;
33 }
```

Figure 4.3: Semple specification of the `vector` `insert` function

CHAPTER 5
Integer analysis

Static analysis for our representation of iterator and container specifications relies greatly on the ability to precisely model the positions of iterators and the sizes of containers. Of paramount importance is the ability to determine the relationship between a nonsingular iterator's position and the size of the container it references, even when neither value can be statically determined. For instance, if the two values are equivalent, the iterator is a past-the-end iterator, whereas if the iterator's position is smaller it is a dereferenceable iterator. A precise integer analysis algorithm will be able to represent the values of integer fields and variables so that these important relationships can be determined statically. Imprecision in the analysis can result in false positives in a static checker such as STLlint.

The higher-level integer analysis introduced in this thesis, called *lazy range propagation*, is a precise symbolic integer analysis that exploits knowledge of inter-object relationships to improve precision. STLlint implements lazy range propagation, permitting it to achieve very low false positive rates and to detect errors in the use of iterators. Chapter 6 builds upon lazy range propagation to enable loop analysis for higher-level induction variables.

This chapter describes our lazy range propagation algorithm in Sec. 5.1, then proceeds to discuss in Sec. 5.2 a situation called *relational overlap*, which degrades the precision of integer analyses based on value range propagation [47] and is of particular importance when analyzing iterators. Sec. 5.3 presents an algorithm capable of eliminating the problems associated with relational overlap in limited cases, which we then generalize to support arbitrary sets of relationships among any number of program variables in Sec. 5.4; Sec. 5.5 applies this algorithm to improve precision for object-oriented programs and specifications.

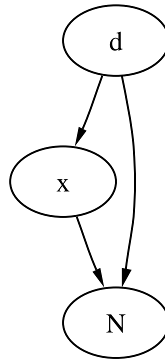


Figure 5.1: The range dependency graph for the expression $d = 1 + (x \in [0 : N - 1]) - N$

5.1 Lazy symbolic range propagation

The lazy symbolic range propagation algorithm builds on Blume and Eigenmann's symbolic range propagation [12], introducing an extra level of symbolic indirection that improves precision. Symbolic range propagation employs a single mapping (called the range dictionary) from program variables to (symbolic) ranges, which may involve other program variables. More importantly, it provides an algorithm for comparing symbolic expressions based on this representation, e.g., to determine if the value of a particular program variable is always less than the value of another program variable. Comparisons are performed by computing the difference between the two expressions and substituting the bounds of symbolic variables for the associated symbolic variables in a meaningful order, until the difference is comparable to zero. For instance, comparing the expression $1 + (x \in [0 : N - 1])$ to N, where $N \geq 1$, performs symbolic variable replacements on the difference $1+(x \in [0 : N-1])-N$ until this difference is comparable to zero. 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 the algorithm to conclude that $1+(x \in [0 : N-1]) \leq N$. Substitution order is determined by the topological sort order of the *range dependency graph* (RDG) [12], a directed graph whose vertices correspond to symbolic variables where an edge (u, v) indicates that the symbolic variable u contains an instance of the symbolic variable v within the value range of u, as is illustrated in

	Program Code	Original formulation	Lazy formulation
1	`if (random) then {`	*maybe-taken*	*maybe-taken*
2	`  x := 3;`	$x = 3$	$x = 3$
3	`  y := 7;`	$y = 7$	$y = 7$
4	`} else {`		
5	`  x := 10;`	$x = 10$	$x = 10$
6	`  y := 15;`	$y = 15$	$y = 15$
7	`}`		
8	`// Join point`	$x = [3 : 10]$, $y = [7 : 15]$	$x = [3 : 10]$, $y = [7 : 15]$
9	`z := x + y`	$z = [10 : 25]$	$z = x + y$
10	`if (x > 5) then {`	$x = [6 : 10]$	$x = [6 : 10]$
11	`  if (z < 13) then`	*maybe-taken*	*never-taken*

Figure 5.2: Example code illustrating one case where lazy symbolic range propagation is more accurate than the original formulation by Blume and Eigenmann

Fig. 5.1.

Assignments to program variables with symbolic range propagation involve calculating the value range on the right-hand side, performing various simplifications to reduce the size of symbolic terms, and updating the range dictionary to map the assigned variable to this new range. For the class of invertible assignments [21], symbolic range propagation offers improved precision relative to the precision it achieves for normal assignments. Invertible assignments are assignments of the form $x_{new} = f(x_{old})$, where the new value for a variable x is dependent on its old value such that one can determine an inverse function f' where $f'(x_{new}) = x_{old}$. Invertible assignments are analyzed by replacing all occurrences of the variable x in the range dictionary with the expression $f'(x)$, then replacing the value of x with x_{new} within the range dictionary.

Lazy symbolic range propagation employs a two-stage mapping. The first mapping associates program variables with symbolic expressions (of the same form as the symbolic bounds) and the second associates the symbols within those expressions with (symbolic) value ranges. In this representation, assignments manipulate only symbolic expressions without performing any value-range–based simplifications. Range substitutions are performed lazily, for expression comparison only, to evalu-

ate boolean relations such as $\leq$ or $>$. Fig. 5.2 illustrates a case where this two-stage mapping can determine that the final `if` conditional will never be true, whereas the original formulation of symbolic range propagation will not be able to determine this. The increase in precision is due to the limited backwards flow of information afforded by lazy range propagation: When the condition `z < 13` is evaluated at line 11, symbolic range propagation uses the range $[10 : 25]$ for z computed at line 9 and determines that the conditional may be true. On the other hand, the two-stage mapping computes the range $[13 : 25]$ for z at line 11, because it is able to incorporate the extra knowledge about x gained at line 10 to narrow the range of z, and therefore the condition `z < 13` can be statically determined to be false.

The control-flow merge operation for lazy symbolic range propagation is roughly equivalent to that of other value range propagation algorithms [12, 47, 84, 115]. In the general case, the merge operation $\phi(x, y)$ produces a new symbolic variable $z \in [\min(x, y) : \max(x, y)]$, although two cases are handled explicitly:

- If x and y cannot be compared, they are both "grounded" to terms involving only integral constants, $-\infty$, or ∞, and the resulting symbolic variable is $z \in [\min(ground(x), ground(y)) : \max(ground(x), ground(y))]$. Although this special case may result in a precision loss, this loss is mitigated by the fact that the expression is only simplifiable when either x or y is later restricted to be comparable with the other.

- If the difference d between y and x is either a positive integral constant or symbolic variable, the result of $\phi(x, y) = x + (w \in [0 : d])$. While this symbolic expression is equivalent to the expression derived from the general case, its form is advantageous in the recognition of monotonic induction variables, described further in Sec. 6.1.3.

5.2 Precision loss due to relational overlap

The control-flow merge operation of value range propagation (and as inherited by lazy range propagation) introduces imprecision into value range propagation that causes the analysis to fail for simple, common programming idioms. Fig. 3.9

illustrates one such idiom: we need only verify that the iterator's position does not exceed the container's size by comparing the value ranges. Within each loop iteration, the loop either erases the current student (decreasing the size of `students`), implicitly setting the iterator to the next student; or explicitly increments the iterator to the next student in the vector. Thus value range propagation can verify in each branch that the iterator remains valid, but once the value ranges have been merged following the if-then-else statement, iterator validity can no longer be proven because the value ranges of the iterator index and the container size will overlap. A *path-sensitive* algorithm, which avoids control-flow merges by following every path through the program separately, will verify iterator validity in Fig. 3.9 whereas a path-insensitive algorithm based on the traditional value range merge operation will not.

To understand the inability of value range propagation to verify iteration validity in Fig. 3.9 with the traditional value range merge operation, we must understand why the merge operation does not preserve arithmetic relations. Let Range denote a set of ranges $[a : b]$ and let variables be drawn from the set Variable. We revisit the general definition of the merge operation applied to value ranges:

Definition 1. *Given a variable v whose value ranges are $[a_1 : b_1], [a_2 : b_2], \ldots, [a_n : b_n]$ on the n incoming control-flow edges, the value range merge operation ϕ : Variable $\rightarrow$ Range is defined as the union of the ranges on each incoming flow edge [47]:*

$$\phi(v) = [\min(a_1, a_2, \ldots, a_n) : \max(b_1, b_2, \ldots, b_n)]. \tag{5.1}$$

We note two important properties of ϕ:

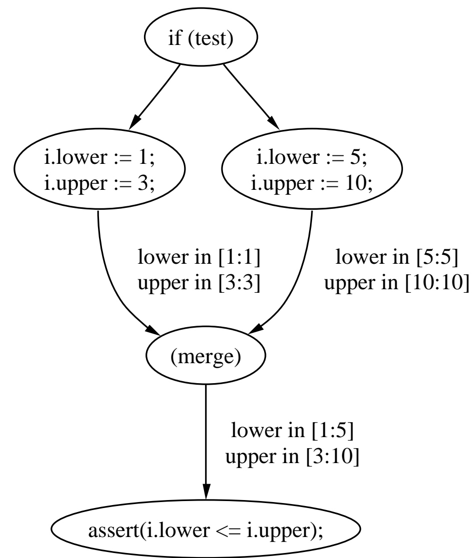
- ϕ is a join operation [20], such that $\phi(v) \supseteq [a_i : b_i]\ \forall_{1 \leq i \leq n}$, meaning that the value range $\phi(v)$ includes the value ranges for v on all incoming flow edges.

- ϕ requires a partial ordering on the value range bounds, but otherwise does not restrict the bounds. Therefore, the value range bounds may comprise simple domains, such as integer or floating-point values [115], or more expressive domains such as symbolic expressions [12, 47].

```
class interval {
    lower :  integer;
    upper :  integer;
}

i :  interval;
if (test) then {
    i.lower := 1;
    i.upper := 3;
} else {
    i.lower := 5;
    i.upper := 10;
}
assert(i.lower <= i.upper);
```

(a) Merging an interval ADT



(b) Control flow graph

Figure 5.3: Merging the value ranges representing the fields in an interval abstract data type results in an unacceptable loss of precision.

Fig. 5.3 illustrates the effects of value range propagation on a very simple program utilizing an interval abstract data type. Each outgoing control-flow edge is annotated with the value ranges for the `lower` and `upper` fields of the interval `i`. For instance, the edge exiting the node corresponding to the `then` branch declares that `lower` $\in [1:1]$ and `upper` $\in [3:3]$ as a result of the two assignment statements in that branch. The edge exiting the `else` branch is similarly annotated. At the control-flow join, marked "merge", the value range propagation algorithm merges the values on each incoming control-flow edge, thus `lower` $\in [\min(1,5) : \max(1 : 5)] = [1:5]$ and `upper` $\in [\min(3,10) : \max(3,10)] = [3:10]$ on the edge exiting the control-flow join.

The value ranges exiting the control-flow join illustrate the loss of precision inherent in the definition of the merge operation. On the edge exiting the `then` and `else` branches, value range propagation can determine that the invariant `i.lower <= i.upper` holds by comparing the upper bound of `i.lower` to the lower bound of `i.upper`. Following the control-flow join, the relation is no longer provable, as

$5 \not\leq 3$: the conservative merge operation dictated by a path-insensitive analysis has degraded the precision such that the analysis can no longer statically determine the correctness of the following `assert` statement. Note that a path-sensitive analysis would not suffer from such an error, as it would in effect have two `assert` statements (one for each branch of the `if` statement).

We label the cause of the analysis' failure to detect the correctness of the `assert` statement as *relational overlap*. Given two variables, l and r, relational overlap occurs when the values of l and r are reassigned in two different control-flow paths such that $l \leq r$ is provable in each path individually but not provable after a control-flow merge. With value range propagation, relational overlap occurs when the merged ranges of l and r overlap (i.e., the intersection of the ranges is nonempty). This overlap occurs regardless of the underlying integral expression domain representing the lower and upper bounds within a value range. Thus, the problem occurs for simple value range bound representations, such as integral constants, as well as very powerful symbolic value ranges. Similar representations, such as sets of disjoint value ranges [84] and lazy symbolic range propagation, exhibit the same behavior.

The interval abstract data type example in Fig. 5.3 illustrates the simplest form of the relational overlap problem. Our motivating example from Fig. 3.9 fails due to relational overlap, but in a more disguised form. Consider the last iteration of the loop in Fig. 3.9, where the position of the iterator `iter` is one less than the size of the container: the `then` branch reduces the size of the container by one and leaves the iterator at the same position, whereas in the `else` branch the size of the container remains the same and the iterator position is incremented by one. In both branches, the position is no larger than the container size, but at the control-flow merge this relation is no longer provable because the iterator position from the `else` branch exceeds the container size from the `then` branch.

Relational overlap occurs in the analysis of simple example programs and in common programming idioms, causing a loss of precision that inhibits verification and optimization. Relational overlap is caused by the value range merge operation ϕ, and cannot be solved directly with more expressive value range representations.

5.3 Relational merging algorithm

Our approach to solving the relational overlap problem is to redefine the value range merge operation to *preserve* relations. A merge operation preserves a relation $x \oplus y$ if the provability of $x \oplus y$ on all incoming control-flow edges implies provability of $x \oplus y$ on the merged result. Sec. 5.2 illustrated that ϕ does not preserve the $\leq$ operation; similar examples show that ϕ does not preserve $<, >, \geq$, or $=$ (equality), either.

Definition 2. *Given variables x and y with value ranges $[a_1 : b_1], [a_2 : b_2], \ldots, [a_n : b_n]$ and $[c_1 : d_1], [c_2 : d_2], \ldots, [c_n : d_n]$, respectively, on the n incoming control-flow edges, we define the merged value range of y* relative to x *as*

$$\phi_{rel}(y, x) = [x + \min(c_1 - b_1, c_2 - b_2, \ldots, c_n - b_n) : x + \max(d_1 - a_1, d_2 - a_2, \ldots, d_n - a_n)] \tag{5.2}$$

The function ϕ_{rel} utilizes the definition of value range comparison to preserve ordering relations across control-flow merges. Following a control-flow merge, the offset of y from x will be compared against zero when x and y are compared. By capturing the offsets between x and y in each control-flow edge at the time of the merge, ϕ_{rel} ensures that the offset precision is not lost due to varying magnitudes of the values of x and y.

Lemma 1. *Given variables x and y, the relative merge operation $\phi_{rel}(y, x)$ preserves the relation $x \oplus y$ for any $\oplus \in \{<, \leq, =, \geq, >\}$.*

Proof. Let x and y be variables with value ranges $[a_1 : b_1], [a_2 : b_2], \ldots, [a_n : b_n]$ and $[c_1 : d_1], [c_2 : d_2], \ldots, [c_n : d_n]$, respectively, on the n incoming control-flow edges. The difference $\phi_{rel}(y, x) - x = [\min(c_1 - b_1, c_2 - b_2, \ldots, c_n - b_n) : \max(d_1 - a_1, d_2 - a_2, \ldots, d_n - a_n)]$ denotes the offset of y from x. The proof follows by examining all possible values of $\oplus$.

1. $x < y$: The relation $x < y$ holds on control-flow edge i if $c_i - b_i > 0$. If $\forall_{1 \leq i \leq n}\ c_i - b_i > 0$, then $\min(c_1 - b_1, c_2 - b_2, \ldots, c_n - b_n) > 0$ and thus $x < y$ in the merged result.

2. $x \leq y$: Analogous to the $x < y$ case.

3. $x > y$: The relation $x > y$ holds on control-flow edge i if $d_i - a_i < 0$. If $\forall_{1 \leq i \leq n}\ d_i - a_i < 0$, then $\max(d_1 - a_1, d_2 - a_2, \ldots, d_n - a_n) < 0$ and thus $x > y$ in the merged result.

4. $x \geq y$: Analogous to the $x > y$ case.

5. $x = y$: The relation $x = y$ holds on control-flow edge i if $c_i - b_i = d_i - a_i = 0$. If $\forall_{1 \leq i \leq n}\ c_i - b_i = d_i - a_i = 0$, then $\min(c_1 - b_1, c_2 - b_2, \ldots, c_n - b_n) = \max(d_1 - a_1, d_2 - a_2, \ldots, d_n - a_n) = 0$ and thus $x = y$ in the merged result.

□

Revisiting Fig. 5.3, we see that merging `upper` relative to `lower` results in `lower` $\in [1 : 5]$ and `upper` $\in$ [`lower` $+ 2 :$ `lower` $+ 5$]. The offset of `upper` from `lower` is clearly positive, so with the revised merge operation the analysis is able to prove statically that the assertion condition is true.

We now generalize relational merging to an arbitrary chain of relations that allows us to increase the precision of relational merging for more than two variables. That is, instead of a single relation $x \oplus y$ with two variables, we consider a chain of $m-1$ relations with m variables $v_1 \oplus_1 v_2 \wedge v_2 \oplus_2 v_3 \wedge \cdots \wedge v_{m-1} \oplus_{m-1} v_m$. We arbitrarily select a value i with $1 \leq i \leq m$, call variable v_i an *anchor point*, and let $v'_i = \phi(v_i)$. From the anchor v_i, we perform cascading relative merges outward, such that the neighbors of v_i, v_{i-1} and v_{i+1}, are merged relative to v_i; the merging continues recursively along the chain until no neighbors exist. More formally, let $1 \leq i \leq n$ and $v'_i = \phi(v_i)$. Then $\forall_{1 \leq j < i}$ we set $v'_j = \phi_{rel}(v_j, v_{j+1})$ and $\forall_{i < k \leq m}$ we set $v'_k = \phi_{rel}(v_k, v_{k-1})$.

Lemma 2. *Given a set of variables $v_1, v_2, \ldots, v_m$ with value ranges on the n incoming control-flow edges, the chain of relations $v_1 \oplus_1 v_2 \wedge v_2 \oplus_2 v_3 \wedge \cdots \wedge v_{m-1} \oplus_{m-1} v_m$ with $\oplus_i \in \{<, \leq, =, \geq, >\}$, and an integer anchor $1 \leq i \leq m$, the cascading chain merge algorithm preserves all relations $v_j \oplus_j v_{j+1}$ for all $1 \leq j < m$.*

Proof. For $1 \leq j < m$, either $v'_j = \phi_{rel}(v_j, v_{j+1})$, if $j < i$, or $v'_j = \phi_{rel}(v_j, v_{j-1})$, if $j > i$. In either case, the relation $v_j \oplus_j v_{j+1}$ is preserved by Lemma 1. □

5.4 Variable relation graph

To accommodate the complex relationships among program variables in real programs, we expand the relational merging algorithm to support an arbitrary set of relations among variables. We represent the set of relationships as edges in an undirected *variable relation graph*, a data structure that serves as the basis for our relational merging algorithm.

We first formalize the notion of the variable relation graph. Given a set of m variables $v_1, v_2, \ldots, v_m \in$ Variable and a set of relations $R \subseteq$ Variable $\times$ Relation $\times$ Variable, where a Relation may be any one of $<, \leq, =, \geq$, or $>$, we construct a *variable relation graph*. The vertices V of the graph are the variables $v_1, v_2, \ldots, v_m$ (we will refer to variables and vertices interchangeably with these names). The undirected edges $E \subseteq$ Variable $\times$ Variable of the graph represent any relationship between two variables. The edge $(v_i, v_j) \in E$ if $i \neq j$ and there exists a $(v_i, \oplus, v_j) \in R$ for some $\oplus \in \{<, \leq, =, \geq, >\}$. The variable relation graph therefore captures the structure of the web of relationships among variables, but not the nature of the relationships.

Definition 3. *Given a variable relation graph $G = (V, E)$ with vertices $V = \{v_1, v_2, \ldots, v_n\}$, let the predicate Preserves(v_i, v_j) return true when $v_i \oplus v_j$ is preserved by the algorithm for $\oplus \in \{<, \leq, =, \geq, >\}$. The precision of a merge algorithm with respect to the variable relation graph G is $\frac{|P|}{|E|}$, where*

$$P = \{(v_i, v_j) | (v_i, v_j) \in E \text{ and } \mathit{Preserves}(v_i, v_j)\}.$$

A more precise algorithm will preserve more relations in the variable relation graph, and therefore will have a higher precision. The precision of ϕ is 0 (it is not guaranteed to preserve any relationships) and the precision of the chain merging operation is 1 according to Lemma 2 if we restrict the variable relation graph to a chain of relations.

```
MergeVisitor =
    start-vertex(v)     {v' = φ(v); }
    tree-edge(u, v)     {v' = φ_rel(v, u); }
```

Figure 5.4: A depth-first search visitor that performs relative merges along a spanning tree in the variable relation graph

5.4.1 Tree properties

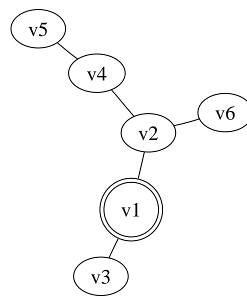
By first restricting the variable relation graph to a tree, we can define a merge operation suitable for variable relation trees with perfect precision. Variable relation trees are unrooted due to their direct connection with sets of variable relations (which have no such concept), so we arbitrarily select a vertex $v_s \in E$ as the root. The root v_s is equivalent to the anchor in the chain-merging algorithm in Sec. 5.3, so we let $v_s' = \phi(v_s)$ and perform cascading merges starting with the root and out to each child node. The cascading merges are performed by executing a depth-first search starting at v_s and applying ϕ_{rel} to the target and source of each tree edge encountered. Fig. 5.4 illustrates the depth-first search visitor that performs the merge for the variable relation tree: the start-vertex event is executed for the starting node of the depth-first search (v_s), and for each tree edge thereafter tree-edge(u, v) is executed as we cross the edge from u to v.

Fig. 5.5 illustrates a variable relation tree and the merges performed by our merge algorithm. The variable relation tree represents the set of relations $R = \{v_1 \leq v_3, v_1 \leq v_2, v_2 \leq v_6, v_4 \leq v_2, v_5 \leq v_4\}$, and we have arbitrarily selected vertex v_1 as the root of the tree. We see that for each tree edge, the target of the tree edge has been merged relative to the source of the tree edge.

Lemma 3. *The merging algorithm for a variable relation tree preserves all relations in the tree. It has a precision of 1.*

Proof. Given a variable relation tree $G = (V, E)$, for each edge $(v_i, v_j) \in E$ the depth-first traversal executes either tree-edge(v_i, v_j) or tree-edge(v_j, v_i), thus performing the relative merge $\phi_{rel}(v_j, v_i)$ or $\phi_{rel}(v_i, v_j)$, respectively. Thus by Lemma 1, all relationships are preserved and the precision is 1. □

The choice of the root of the tree, v_s, is arbitrary and does not affect the



(a) Variable relation tree

$$\begin{aligned} v_1' &= \phi(v_1) \\ v_2' &= \phi_{rel}(v_2, v_1) \\ v_4' &= \phi_{rel}(v_4, v_2) \\ v_5' &= \phi_{rel}(v_5, v_4) \\ v_6' &= \phi_{rel}(v_6, v_2) \\ v_3' &= \phi_{rel}(v_3, v_1) \end{aligned}$$

(b) Merges performed on variables in the tree

Figure 5.5: A variable relation tree where vertex v_1 is arbitrarily selected as the root

asymptotic behavior of the algorithm. However, depending on the domain of the value range bounds, it may affect the overall precision of the analysis. Paths through the variable relation tree could theoretically be very long, creating a long chain of dependencies. Calculating the actual value range requires a substitution step for each edge in the path, and most symbolic analyses limit the number of substitution steps to avoid infinite recursion. Once the substitution limit has been reached, an analysis substitutes the most conservative bound, e.g., $[-\infty : \infty]$ or $\top$, reducing the precision of the resulting value. For this reason, the tree center is a good choice for the root vertex v_s, because it minimizes the number of substitutions that will need to be performed.

The depth-first visitor formulation of the merging algorithm in Fig. 5.4 applies equally well to a variable relation forest. The depth-first search executes start-vertex for each vertex that is not reachable from any edge of an existing tree in the forest, and therefore effectively partitions the forest into separate trees merged independently.

5.4.2 Graph properties

A depth-first traversal of a graph crosses the tree edges of a spanning tree in each connected component within the graph. We therefore construct our merge algorithm for variable relation graphs in two phases: the first phase selects a spanning

tree in each connected component of the graph, and the second phase applies the depth-first–search visitor in Fig. 5.4 to each of these spanning trees.

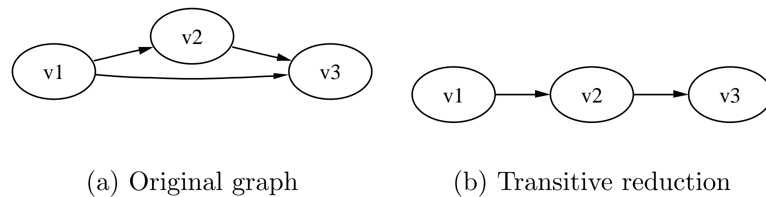
The occurrence of cycles in the variable relation graph $G = (V, E)$ results in a loss of precision of the merge algorithm. Given two paths from a vertex v_i to another vertex v_j, the relations on both paths cannot be guaranteed to be satisfied, as at most one of the paths will become a part of the spanning tree. Thus the relations for edges not part of the spanning tree are not guaranteed to hold following the control-flow merge. The set of edges P whose relations are guaranteed to be preserved is equivalent to the set of edges in the spanning tree (or spanning forest). The precision of the algorithm is therefore $\frac{|P|}{|E|} = \frac{|V|-1}{|E|}$.

5.4.3 Spanning tree selection and pruning heuristics

The spanning tree selection in the first stage of the merge algorithm for the variable relation graph determines the set of relations that are preserved by the algorithm. We would prefer to select edges for relations that are more important for the goals of the analysis: for instance, an edge representing an invariant relation (i.e., one that is always true) may be more important than an edge representing a relation that would only benefit local optimizations. Similarly, the likelihood of a comparison actually being performed may weigh into the importance of an edge: one can count the static or dynamic occurrences of comparisons mapping to a particular edge as a measure of the importance of that edge. Once the application-specific importance measures have been coalesced, a maximal spanning tree algorithm may be used to select a spanning tree to be merged.

We may also prune unnecessary edges from the variable relation graph prior to spanning tree selection. We label a relation *implied* when the relation is a logical consequence of other relations, and therefore can be removed from the set of relations without affecting the final results. By analogy, an implied edge is an edge that represents an implied relation. Removing an implied edge from the variable relation graph increases the precision of the merge algorithm.

Implied edge candidates can be identified in a directed form of the variable relation graph. In this directed variable relation graph, an edge (v_i, v_j) denotes the



(a) Original graph (b) Transitive reduction

Figure 5.6: The transitive reduction of a directed variable relation graph may reduce the effective precision of the merge operation.

relation $v_i \leq v_j$. Thus when there exists an edge (v_i, v_j) and another path from v_i to v_j (that does not contain (v_i, v_j)), the edge (v_i, v_j) may be implied. The variable relation graph, however, does not distinguish between relations that are invariant and those that are *transient*, i.e., may not hold in all paths at a particular control-flow merge. If the path from v_i to v_j contains only edges representing invariant relations, the edge (v_i, v_j) is implied and may be removed. Consider the directed variable relation graph and its transitive reduction in Fig. 5.6: if the relations (v_1, v_2) and (v_2, v_3) are invariant, then they must both be true at any program point and, therefore, (v_1, v_3) is implied and may safely be removed as in the transitive reduction of the graph. However, if the edge (v_1, v_2) is a transient relation, then it is possible that the relations represented by (v_2, v_3) and (v_1, v_3) hold on all incoming paths, but (v_1, v_2) does not; then our pruning will have made it impossible for the merge algorithm to guarantee the preservation of the relation represented by (v_1, v_3). Thus when the variable relation graph (or parts of it) represent intended program invariants, precision can be improved by computing the transitive reduction of the directed variable relation graph and removing the edge directions.

STLlint implements heuristics counting both static and dynamic occurrences of comparisons between nodes in the field relation graph (FRG), discussed in Sec. 5.5, to aid in the selection of root nodes, although spanning tree selection is random. Dynamic occurrences are counted based on the full dynamic FRG, although only the portion of the FRG containing modified fields is actually constructed. Elimination of implied edges is not implemented in STLlint due to the lack of class invariant detection or annotations.

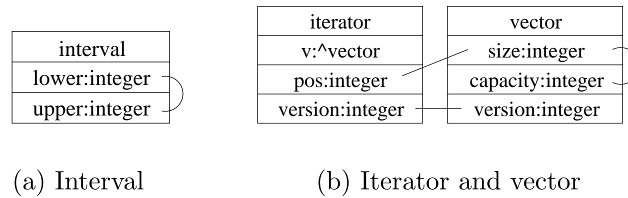
5.5 Merging objects

The relational merge algorithm discussed in Sec. 5.4 depends on the construction of the variable relation graph. Thus far, we have assumed that variables correspond to program variables. While this assumption may be suitable for low-level internal representations that primarily consider integral or floating-point registers, it is not a particularly good assumption for higher-level internal representations, such as those used in the analysis of object-oriented programs.

In object-oriented programs, the majority of the important transient and invariant relations relate fields within an object or fields between objects. The example in Fig. 5.3 illustrates the most basic *intra*-object dependency, where a relation exists between two fields within the same object, e.g., `i`.*lower* $\leq$ `i`.*upper* for object `i`, and Fig. 3.9 illustrates *inter*-object dependencies, where an important relation exists between a field of one object and a field of another object, e.g., `iter`.*pos* $\leq$ `students`.*size*.

We introduce two *field relation* graphs (FRGs) that capture the relationships between fields. The field relation graph is similar to the role reference diagram in role analysis [65] because it captures relationships between fields of different objects, although the form of the information is different. There are two types of FRGs: the *static* field relation graph (discussed in Sec. 5.5.1), which describes relations between the fields of classes, such as the relationship between an iterator's position and a container's size, and the *dynamic* field relation graph (discussed in Sec. 5.5.3), which describes relations between fields of particular objects, such as the relationship between `iter.`*pos* and `students.`*size* in Fig. 3.9.

The two FRGs are directly related because they describe the same properties, i.e., relationships between fields, but we find that the static FRG is more readily constructed given a representation of the program source code (see Sec. 5.5.2), whereas the dynamic FRG is more useful for program analysis. In Sec. 5.5.4, we discuss a transformation that constructs a dynamic FRG from a static FRG, a process we call *instantiation*, drawing on the notion of access paths [67] to provide the crucial link between the two field relation graphs.



(a) Interval (b) Iterator and vector

Figure 5.7: Static field relation graphs for the specification of the interval, iterator, and vector abstract data types

5.5.1 Static field relation graph

The static field relation graph captures the relationships between fields based on the class types involved. For instance, within the abstract data type `interval` of Fig. 5.3 there exists the relationship `interval`.*lower* $\leq$ `interval`.*upper*, which must hold for any object of type `interval`. Similarly, there may exist dependencies between fields in different class types, e.g., `iterator`.*pos* $\leq$ `vector`.*size*, as in Fig. 3.9. Fig. 5.7 illustrates two static field relation graphs. The first graph contains an *intra*-class relationship, i.e., a relationship that occurs between two fields within the same object, between the *lower* and *upper* fields of an `interval` type. The second graph contains an intra-class relationship between the *size* and *capacity* fields of a `vector` specification, but also an *inter*-class relationship, i.e., one that relates two fields from different object types, between the iterator's *pos* field and the vector's *size*. Like the variable relation graph, this graph is undirected and therefore describes only the fact that a relationship exists and not the direction of the inequality.

The nodes in the static field relation graph are fields of a particular class type and are written as $C.m$, where C is the class type and m is a field in that class. The undirected edges $(C_1.m_1, C_2.m_2)$ represent a relationship between field m_1 of some object o_1 of type C_1 and field m_2 of some object o_2 of type C_2.

The static relation graph can be constructed with information from several sources; it is discussed in Sec. 5.5.2. The graph can be used as input to the graph-based relational merging algorithm when objects of each class type can be trivially assigned. This occurs in several cases:

- When the analysis merges the states of all objects of a particular type into a

single object of that type. In this case, objects are synonymous with classes, and the analysis proceeds as if fields were not instance-specific.

- When a connected component of the graph is restricted to fields within a single class type and the edges of the component relate to fields that will only be compared within a particular object, i.e., it is guaranteed that for the edge $(T.m_1, T.m_2)$ and objects o_1 and o_2 of type T, a comparison between $o_1.m_1$ and $o_2.m_2$ requires $o_1 = o_2$. Sec. 5.5.2 further discusses detection of this case.

5.5.2 Computing the static field relation graph

The static field relation graph can be computed with a flow-insensitive, context-insensitive analysis of a representation of the program source code or user annotations. Field relationships come from comparisons between fields in the program source, e.g., a comparison `a.`m<`b.`n that appears in the source code generates an edge between field m in the class type of object `a` and field n in the class type of object `b`. There are several ways that relations may be found in the program source representation, each of which provides slightly different semantic information:

- Relations may actually be user-specified class-level invariants. In this case, the comparison itself is implicitly represented by the existence of the invariant. Edges in the static field relation graph generated from these invariants are useful for the relation graph pruning discussed in Sec. 5.4.3.

- Relations may come from assertion, precondition, or postcondition statements in the program code. The edges generated from these comparisons are locally invariant, because they are guaranteed to hold at the point of the assertion, precondition, or postcondition in a correct program.

- Relations may come from arbitrary conditionals in code, such as assignments to boolean variables or branch conditions. These edges detect only *controlling* relational properties, e.g., properties that affect the control flow of the program but not necessarily its correctness. Nonetheless, they do prove useful for enhancing the precision of analysis when the properties can be exploited locally.

During the construction of the static field relation graph, two types of edges are considered: intra-object relation edges and inter-object relation edges. An inter-object relation edge is an edge from a particular field in an object of some type to a field in an object of another (possibly different) type. An intra-object relation edge is an edge from a particular field in an object to another field in that same object. We note that an intra-object edge is an inter-object edge, so an analysis may conservatively classify all edges as inter-object. A comparison between fields within the program source code generates an intra-object edge when it is statically guaranteed that the two fields occur in the same object, e.g., the fields are accessed via the same object name as `o.`m and `o.`n.

When the static field relation graph contains a connected component consisting only of intra-object edges, these subgraphs of the static field relation graph can be used directly with the graph-based relational merge algorithm by applying the algorithm to the subgraphs for each object of the class type referenced by the subgraph. Moreover, because all of the information is static, graph pruning and spanning tree selection can be performed only once off-line.

5.5.3 Dynamic field relation graph

The static field relation graph captures relationships among fields in different class types, but fails to differentiate between objects that have the same class type. For instance, in Fig. 3.9, the `students` and `fail` vectors have the same type, but there are no interdependencies between them because the two are distinct objects. Perhaps more importantly, iterators have dependencies only with the containers they reference but not with other containers of the same type. The distinction is crucial to the verification of the sample code: the `push_back` operation may invalidate iterators referencing the underlying vector. Thus, an analysis that does not differentiate between iterators that reference the vector `students` and iterators that reference the vector `fail` cannot prove that the iterators that reference `students` are not invalidated by a call to `fail.push_back`.

The dynamic field relation graph captures the relationships between instance-specific fields of objects, disentangling dependencies that appear to occur at the type

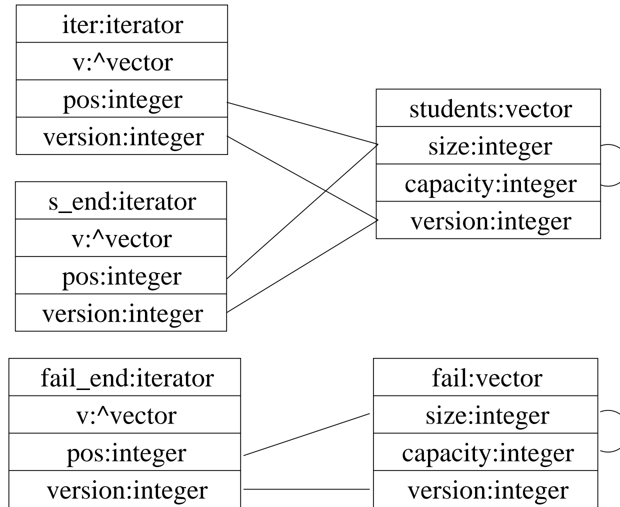


Figure 5.8: Dynamic field relation graph for the objects in Fig. 3.9

level but do not occur at the object level, such as the relationships between iterators and the containers they reference. Fig. 5.8 illustrates the dynamic field graph for the important objects in our motivating example from Fig. 3.9, and here we see that the connected components for the `fail` and `students` vectors are distinct, even though the two vectors have precisely the same type. Thus the dynamic field relation graph can accurately model the relationships amongst objects in a program.

The dynamic field relation graph may be used directly with the graph-based relational merge algorithm. The value of a particular field in an object constitutes a variable in the variable relation graph, so the relational merge considers the edges in the dynamic field relation graph as edges in the variable relation graph, and therefore performs a control-flow merge operation that preserves important intra- and inter-object relationships. It is this formulation that provides the necessary framework to prove iterator validity in Fig. 3.9: the dynamic field relation graph provides an accurate model of the inter-object field relationships at each control-flow join and the relational merge algorithm preserves all relationships within the dynamic field relation graph, because in this case the graph (shown in Fig. 5.8) is a forest for which the relation merge algorithm guarantees perfect precision.

5.5.4 Instantiation

While the static field relation graph can be constructed from the program source code, the dynamic FRG cannot be synthesized so readily, because it must account for object references at each particular program point where the graph may be needed. However, analysis of our motivating example requires the precision of the dynamic FRG and therefore we describe an efficient algorithm for the construction of the dynamic FRG given an approximation of the references between objects as represented in a storage shape graph [18] and the static FRG, a process we refer to as *instantiation*. The storage shape graph, also called a points-to graph, contains a node for each abstract location in the program and (directed) edges from pointer nodes to the nodes that the pointers reference.

The naïve instantiation algorithm assumes that for every edge $(C_1.m_1, C_2.m_2)$ in the static FRG, every object o_1 of type C_1, and every object o_2 of type C_2 there is a relationship between $o_1.m_1$ and $o_2.m_2$. This algorithm produces very dense dynamic field relation graphs, in the worst case producing a complete graph when given objects of a single class type. The resulting dynamic FRG will relate many objects that may otherwise be mutually unreachable in the storage shape graph. For instance, in addition to the edges in Fig. 5.8, the dynamic field relation graph produced by the naïve instantiation algorithm would include edges relating iterators from different containers and iterators with containers they cannot access. The naïve instantiation algorithm therefore reduces the precision of the relational merge algorithm by drastically increasing the number of edges in the graph. The reduced precision results in ineffective relational merging, because the important relations in the graph are not guaranteed to be preserved and the algorithm is no longer able to solve real problems, such as the verification of Fig. 3.9.

The goal of our instantiation algorithm is to capture only the relationships that can be useful at run time. To achieve this goal we determine the ways by which fields of different objects are compared. For instance, an iterator is considered valid if its position (`iter.`*pos*) does not exceed the size of the vector it references, which is stored in the object field `iter.`*v*. Thus, the real dependency is between `iter.`*pos* and `iter.`*v.size*, instead of between `iter.`*pos* and `vec.`*size* for some arbitrary vector

object `vec`.

For each $(C_1.m_1, C_2.m_2)$ in the static field relation graph, we record the *access paths* α_1 and α_2 that result in the member accesses $C_1.m_1$ and $C_2.m_2$, respectively. An access path [67] is an expression mapping from an object through a (possibly empty) sequence of object field references and eventually to a field. Thus $o.m_1.m_2$ is an access path expression that begins with object o, projects to the object referenced by field m_1 in object o, and then projects to field m_2 of that object. The simplest form of an access path expression only follows field references, although for any particular language, access paths can include other expressions, such as type casts, pointer dereferencing, or even pointer arithmetic. We will focus only on access path expressions involving field references.

Given access paths α_1 and α_2 that produce an edge $(C_1.m_1, C_2.m_2)$ in the static field relation graph, we calculate the longest common prefix α_p of α_1 and α_2 and note the class type C_p of the object resulting from the evaluation of the partial access path α_p. The class type C_p *generates* the edge $(C_1.m_1, C_2.m_2)$ if from some object of type C_p in the program we can establish a relationship between $C_1.m_1$ and $C_2.m_2$. We let β_1 and β_2 be the rest of the access paths that originate at objects of class type C_p, such that $\alpha_1 = \alpha_p.\beta_1$ and $\alpha_2 = \alpha_p.\beta_2$. We record the access path pair (β_1, β_2) in a list of generated edges for class type C_p. The access path pair will later be used to construct edges within the dynamic FRG. This computation is performed off-line, and the result is a set of access path pairs for each class type. We note that when $C_1 = C_2$, $\beta_1 = m_1$, and $\beta_2 = m_2$ the edge $(C_1.m_1, C_2.m_2)$ is an intra-object dependency; otherwise, the edge is an inter-object dependency.

Given these access path pairs for each class type, we construct the edges of the dynamic FRG by following access path pairs from any object of the originating type. For each object o with class type C_p, we traverse the list of generated edges for that type C_p. At each access path pair (β_1, β_2), we traverse the access paths starting with object o (e.g., by following edges in the storage shape graph) to determine the two vertices in the dynamic field relation graph that may be related because they could be compared via the object o. We then add an edge between the two vertices in the dynamic field relation graph. This step is computed prior to a relational merge

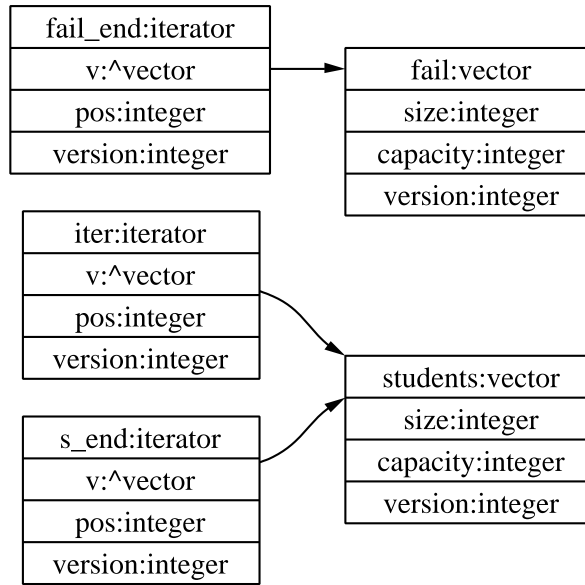


Figure 5.9: Points-to graph for the loop in Fig. 3.9

operation when a flow-sensitive storage shape graph is used, or can be computed once off-line if a flow-insensitive storage shape graph is available.

For the iterator validation example in Fig. 3.9, the access paths for the comparison `iter.`*pos* $\leq$ `iter.`*v.size* are $\alpha_1 =$ `iter.`*pos* and $\alpha_2 =$ `iter.`*v.size*. The longest common prefix is $\alpha_p =$ `iter` with class type $C_p =$ `iterator`. Then $\beta_1 = pos$ and $\beta_2 = v.size$, and the access path pair (*pos*, *v.size*) is added to the set of edges generated by `iterator`. When the dynamic field relation graph is instantiated with the storage shape graph (given in Fig. 5.9), we consider the access path pair (*pos*, *v.size*) for all objects of type `iterator`, and will reconstruct all edges involving an iterator's *pos* field and a vector's *size* field that appear in Fig. 5.8, without creating any additional edges. For instance, given the iterator variable `iter`, whose field v is a pointer to the `students` vector, we follow the access path $beta_1 = pos$ to the location `iter`.*pos* and follow the access path $\beta_2 = v.size$ to the location `students`.*size*, creating an edge between these two nodes. The edges between *version* fields are generated analogously, completing the dynamic FRG for the example.

The instantiation algorithm described here efficiently constructs a precise dynamic field relation graph from a storage shape graph and the static field relation graph. The algorithm complexity is $\mathcal{O}(|O| \cdot |G| \cdot p)$, where $|O|$ is the number of

objects (locations in the storage shape graph) whose type may generate any edges in the dynamic FRG, $|G|$ is the maximum number of access path pairs that generate edges from any particular class type, and p is the maximum length of any access path. In real programs, the access path length and the number of edges generated from any particular class type tend to be rather small, and in practice they are bound to small constants.

5.5.5 Pruning the dynamic field relation graph

The dynamic field relation graph for a particular program may become very large, as it contains all intra- and inter-object relationships that may affect the analysis. For a particular control-flow join, however, it is rare that the values differ across incoming control-flow edges for all fields in the graph, with many control-flow joins affecting only a small subset of the fields in the graph. Pruning is accomplished by constructing vertices only for field instances where the value differs on any two incoming control-flow edges; edges that would be introduced into the graph are only added if both vertices represent such fields, otherwise they are discarded.

The dynamic field relation graph that results from pruning during instantiation retains all relationships important to the static analysis that may not be preserved during a control-flow merge, but eliminates relationships that cannot change due to the control-flow merge. Pruning eliminates the cost of merging values that do not differ along the incoming control-flow edges and can drastically reduce the size of the dynamic FRG. Moreover, connectivity-based statistics for the full dynamic FRG, which are useful for spanning tree selection, can be calculated without building the full FRG, so that spanning tree selection heuristics do not suffer from pruning.

5.6 Algorithm complexity

Our lazy symbolic range propagation algorithm exhibits the same overall complexity as Blume and Eigenmann's symbolic range propagation. The most expensive operation is symbolic expression comparison, which requires $\mathcal{O}(|V|^2)$ variable replacements in the worst case. However, while the size of the variable set $|V|$ in symbolic range propagation is equal to the number of program variables (and

therefore bounded by program size), lazy range propagation introduces variables at control-flow merges (potentially one for each program variable), and is therefore bounded by the square of the program size. On the other hand, the complexity of performing assignments is somewhat lessened by lazy range propagation, because invertible assignments do not require modification of every value in the range dictionary. The simplification work is delayed until expression comparison, where the increase in $|V|$ increases the complexity of lazy symbolic range propagation relative to symbolic range propagation.

The algorithms presented for construction of the dynamic field relation graph and for relation-preserving value merging are executed once for each control-flow merge during the static analysis of a program. This frequency of execution requires that the algorithms be efficient in both time and space, and that they scale well with program size. Application of these algorithms occurs in two stages: construction of the dynamic FRG and merging along the edges of a spanning tree in the dynamic FRG.

The first stage, construction of the dynamic FRG as discussed in Sec. 5.5.4, requires iteration over each of the objects that can generate edges in the dynamic FRG. The number of objects that can generate edges is bounded by the number of objects in the abstract program state, $|O|$. We let $|P|$ denote the maximum number of access path pairs generated by any type. For any particular access path pair, computing the field instance referenced by the access path from a particular object is linear in the length of the access path, although we ignore this effect by bounding the maximum access path length to some arbitrary constant. Constructing the dynamic FRG therefore considers $\mathcal{O}(|O| \cdot |P|)$ edges, $|O|$ vertices, and executes in $\mathcal{O}(|O| \cdot |P|)$ time.

The second stage, which includes spanning tree selection and relation-preserving merging, traverses all edges in the dynamic FRG and applies either the merge operator ϕ (Def. 1) or the relative merge operator ϕ_{rel} (Def. 2) at each vertex based on the spanning-tree traversal. We denote the execution time complexity of ϕ and ϕ_{rel} by $\mathcal{O}(\phi)$ and do not specify it further because it differs greatly among different representations: very simple value range propagation, for instance, may require time

proportional to the number of control-flow edges, whereas symbol range propagation may require cubic time in the number of program variables for each flow edge in the worst case [12]. The merge operation is applied to each field of each object, and we let $|F|$ denote the maximal number of fields in any object type. The cost of relational merging along a depth-first traversal of the dynamic FRG is therefore $\mathcal{O}(|O| \cdot |P|) + |O| \cdot |F| \cdot \mathcal{O}(\phi)$.

The algorithmic complexity for $|O|$ objects that generate $|P|$ access path pairs each is therefore $\mathcal{O}(|O| \cdot |P| + |O| \cdot |F| \cdot \phi)$. The sizes $|O|$, $|P|$, and $|F|$ are bounded by the program size, implying a quadratic number of applications of ϕ in the worst case. Within the STLlint test suite (described in Sec. 8.4), $|P|$ is never larger than 5, and is expected to be bounded by a small constant in other applications.

5.7 Summary

The integer analysis employed for higher-level static analysis is lazy symbolic range propagation, a derivative of symbolic range propagation. Lazy symbolic range propagation introduces an additional level of indirection that results in more precise analysis of integer values and integer expression comparisons. Moreover, this formulation enables symbolic loop analysis, discussed in Chap. 6.

The precision of value range propagation, and in particular lazy symbolic range propagation, can be improved via the relational merging algorithm presented here. The relational merging algorithm, when applied to the field relation graph, preserves important interobject relationships for the analysis and checking of programs such as that of Fig. 3.9.

CHAPTER 6
Loop analysis

The greatest challenge in lifting static analysis to library-level analysis has revolved around loop analysis. 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, and induction variables accessed/modified via (multi-level) pointers. Also, 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. The deceptively simple loop in the subroutine `extract_fails` of Fig. 3.9 (page 47) exhibits all of these characteristics when analyzed using the iterator specifications described in Sec. 3.3.1:

- Within the loop, the primary induction variable is the iterator `iter`, which steps through the `students` vector via the `++` prefix operator, which increments the position of the iterator by one step, and via the `erase` operation, which in effect leaves the position of the `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 (higher-level) monotonic induction variable, because its position moves either zero or one steps forward in each iteration.

- The expression `++iter` is a call to a subroutine 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 [15, 20] 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 `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, there are many precondition 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 operation requires that the *position* of the iterator be less than the size of the *sequence* it references. 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. 3.9, brings us back to the monotonically nonincreasing size of the `students` vector.

This chapter discusses a variety of techniques to analyze iterator loops within a unified symbolic execution framework. In the following sections, we focus on the analysis of integer values within loops, 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. STLlint implements the analyses presented in this chapter, allowing it to determine precise information about the movements of iterators within program loops.

6.1 Induction variable recognition

The representation of integer values as symbolic expressions is crucial to the application of induction variable recognition. As demonstrated in Fig. 3.9, 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.

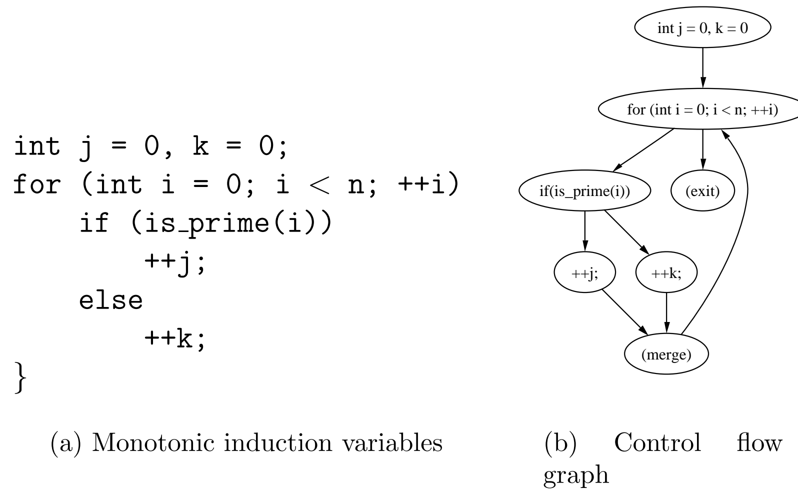
6.1.1 Symbolic differencing

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 [2, 121], most notably those that are based on Static Single Assignment (SSA) form [4, 88], are unusable in this context because one cannot associate every abstract location with a program variable. For this reason, we employ symbolic differencing [45, 46] to recognize the inductive behavior of integer locations. 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 of degree m with loop-invariant coefficients and loop-invariant a and r, where the maximal degree of the polynomial, m, is a parameter to the symbolic differencing analysis that defaults to the value 5. 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 $m+2$ 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, we can derive induction expressions for integer locations regardless of how—or where—they are accessed. Thus in an iterator loop such as that of Fig 3.1, the location `first.`*position* will originally be assigned a fresh symbolic variable p, and will assume the values $p+1$, $p+2$, ..., $p+m+2$ at the end of the $m+2$ loops. Applying symbolic differencing to this sequence $(p, p+1, p+2, \ldots, p+m+2)$, 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.

6.1.2 Trip count calculation

The generality of symbolic differencing also enables accurate loop trip count [45, 92, 109] determination, i.e., computing 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. 3.1, for instance, the first portion of the loop termination condition `first != last` occurs roughly in the middle of each loop iteration when subroutines are considered; see Fig. 6.2. When this condition evaluates true, the second portion of the loop termination condition, `!pred(*first)`, dereferences the `first` iterator and executes an arbitrary subroutine provided by the caller, 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 referring to the underlying integer locations. We utilize the latter option, by computing the set of integer expressions involved in the 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



(a) Monotonic induction variables

(b) Control flow graph

Figure 6.1: A simple loop containing a basic induction variable i and two monotonic induction variables, j and k

expressions when they occur within subroutines are described in Sec. 6.2.1. 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 [46].

6.1.3 Monotonic induction variables

The introduction and use of value ranges at merge points in traditional value range propagation 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. 6.1 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. 6.1 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 $j_x + \alpha$. 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 $j_x, j_x + \alpha, j_x + 2\alpha, j_x + 3\alpha, j_x + 4\alpha$. This behavior, noted but not discussed in Sec. 5.1, is employed for all merge points, where α is unique to the control-flow join, abstract location being merged, and call context. 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 given any (symbolic) n, we can determine that following execution of the loop, the value of j will be its initial value j_0 plus αn, i.e., some value between $[0 : n]$.

The iterator-erasure example in Fig. 3.9 (page 47) illustrates an interesting case where two monotonic variables—`iter.`*position* and `students.`*size*—are in fact related, because only one of these variables will change in any given iteration. This information is crucial to the proper analysis of the example, where the invariant `iter.`*position* $\leq$ `students.`*size* must be proven true. In fact, the relational merging algorithm described in Chap. 5 preserves this relation, and the induction expressions computed for these values capture both their monotonicity and their interdependence, allowing STLlint to verify the correctness of the example.

6.2 Interprocedural loop analysis

We address 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 (Chap. 5) and applying symbolic differencing to these locations (Sec. 6.1.1), has already been discussed. Coupled with an appropriate context-sensitive points-to analysis, our approach to loop analysis 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. 6.2.1, and also benefit from techniques that simplify the loop termination condition, as discussed in Sec. 6.2.2.

6.2.1 Interprocedural trip count calculation

The trip count calculation described in Sec. 6.1.2 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 (i.e., a call string [100]). Thus if the 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. We represent these integer expressions as $(context, expression)$ pairs, where the *context* refers to the node within the invocation graph [27] 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 efficient representation and comparison of call strings.

Fig. 6.2 contains a partially-expanded implementation of the `find_if` algorithm originally presented in Fig. 3.1, 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`, which itself is the result of a call to the unknown predicate `pred`. Denoting call strings via $s_1 : s_2 : \cdots : s_n$, our true loop termination condition is

$$(\texttt{find_if} : \texttt{not_equal}, \texttt{x->}\mathit{position}) \neq (\texttt{find_if} : \texttt{not_equal}, \texttt{y->}\mathit{position}) \wedge \mathit{unknown}.$$

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. We construct the loop termination condition by computing an SSA numbering [4, 88] for each subroutine, so that boolean variables occurring

```
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)
{
  assert(x->sequence != NULL
         && x->position <= x->sequence->position);
  assert(y->sequence != NULL
         && y->position <= y->sequence->position);
  assert(x->sequence == y->sequence);
  return x->position != y->position;
}
```

Figure 6.2: Partial expansion of the `find_if` function from Fig. 3.1, illustrating the static analyzer's view of the "simple" iterator loop

in the termination condition may be (recursively) replaced with the appropriate definition. The replacement continues until a recursive call is found or an arbitrary replacement limit is reached. Variables whose definition is the result of a function call will be replaced with the return expression within the callee 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 $m + 2$ 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 result in an exact trip count.

- Comparisons between two linear induction expressions result 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. 6.2, given the initial value 0 for (`find_if` : `not_equal`, `x->`*position*) and N for the initial value of (`find_if` : `not_equal`, `y->`*position*), will be computed as $\min(N, [0 : \infty]) = [0 : N]$.

6.2.2 Goal-directed inlining

We perform interprocedural loop trip count determination to accurately analyze the behavior of program loops with higher-level iteration constructs, allowing us to compute trip counts when many other static analyses could not. Another problem that plagues static analyzers when faced with such loops is that flow-sensitive

```
do {
  iterator_increment(&first);
  assert(first.sequence != NULL
         && first.position <= first.sequence->position);
  assert(last.sequence != NULL
         && last.position <= last.sequence->position);
  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 6.3: "Optimized" form of the `do–while` loop from Fig. 6.2, after boolean variable definitions have been "pulled" into the `if` condition

information generated from conditional branches is not easily propagated. For instance, in Fig. 6.2 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. At the same time, it is essential that this 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, we perform 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. The process is performed from the bottom of the call graph upwards, allowing conditions buried more than one level deep in the call stack

to be pulled into conditional branches above. Fig. 6.3 illustrates the result of pulling the conditional expression for `b4` into the `find_if` loop; we see that this result enables the propagation of 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.

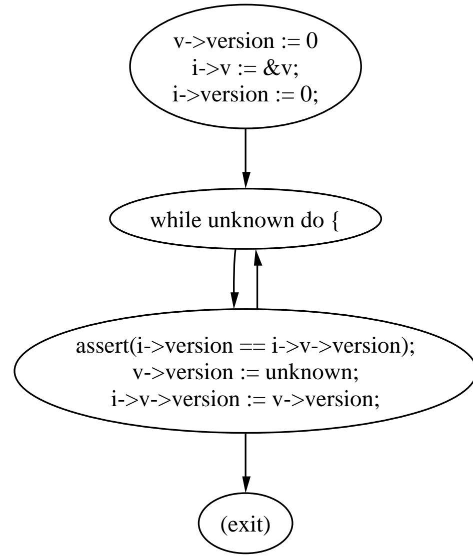
6.3 Proving noninductive equivalence

The techniques presented previously are sufficient to recognize and describe many forms of inductive variables. However, not all interesting integer values within loops are recognizable as inductive, and therefore we must conservatively assume that the values at these locations fall within the unbounded range $[-\infty : +\infty]$. Although we cannot recover the precise values at these locations, the values themselves are often not necessary: often we need only retain information about relationships amongst different abstract locations. With higher-level analysis of the C++ STL, iterator and container version numbers present the most compelling illustration of the need to retain information about equality relationships. Container version numbers, particularly those of a `vector` or `deque`, change in ways that are particularly hard to recognize as inductive, owing to the complex semantics of iterator invalidation.

Fig. 6.4 provides a simplified program illustrating the problem of determining noninductive equivalence. The *version* fields of both the vector `v` and iterator `i` must coincide throughout all iterations of the loop for the assertion to prove true. However, the assignment from `unknown` eliminates the potential for recognizing the version numbers as inductive, because the conservative approximation must assign distinct unbounded values to distinct noninductive locations. The solution here is to perform an inductive proof of the equality of the integer locations corresponding to `i->`*version* and `v->`*version*. The basis case of the inductive proof is readily verified by confirming equality of the values at the two locations in the loop preheader; the inductive case requires that we replace the values at both locations with a fresh symbolic value $x \in [-\infty : +\infty]$, execute the loop body under the most general conditions, and verify that the values at both abstract locations are equivalent at the end of the iteration.

```
while unknown do {
  assert(i->version == i->v->version);
  v->version := unknown;
  i->v->version := v->version;
}
assert(i->version == i->v->version);
```

(a) Noninductive equivalence classes



(b) Control flow graph

Figure 6.4: A loop containing two noninductive but equivalent values, `i->version` and `i->v->version`

This algorithm is easily extended to multiple equivalence classes of arbitrary size. In the basis case, we define equivalence classes $e_1, e_2, \ldots, e_n$ containing variables whose values are equal in the loop preheader. For the inductive case, we construct fresh symbolic constants $x_i \in [-\infty : +\infty]$ for $1 \leq i \leq n$ and set the value of each location $l \in e_i$ to x_i. The loop body is then symbolically executed under the most general conditions, and at the end of the loop body we verify the partitioning of the equivalence classes. If for any pair of locations $l_1, l_2 \in e_i$ the relation $l_1 = l_2$ is not provable, we eliminate e_i from the set of equivalence classes and construct new, maximal equivalence classes from the locations in e_i based on the values stored at these locations. If any equivalence class has been eliminated, we repeat the entire process by constructing fresh symbolic constants for each equivalence class and executing the loop body again. Termination is guaranteed by the strictly decreasing average size of the equivalence classes in each iteration until a fixed point is attained, where all equivalence classes are consistent.

The initial set of equivalence classes may be constructed given the abstract

program state within the loop preheader by building maximal equivalence classes from the values of integer variables, in essence constructing the most eager basis case in the inductive proof. Naïve application of this construction results in $\mathcal{O}(n^2)$ integer comparisons, where n is the number of integer locations in the program state, and is therefore impractical. Moreover, the cost of constructing overly-eager equivalence classes is high, as it requires additional iterations to stabilize. We therefore limit the initial equivalence classes to equivalence relations present within the dynamic field relation graph constructed at the loop preheader. Thus we select equivalence classes that have a high probability of being retained and that also are considered important to the analysis, as we have done with relational merging in Sec. 5.3.

6.4 Loop analysis algorithm

High-level loop analysis, particularly the recognition of higher-level induction variables, requires new techniques for interprocedural loop analysis, recognition of monotonic variables, interprocedural trip count determination, and the computation of noninductive equivalence classes. Although these analyses are presented separately, the need for an efficient implementation compels us to minimize the number of times the loop body must be executed during loop analysis. We propose, and have implemented in STLlint, a loop analysis algorithm that performs the following steps for each loop:

1. Determine the set of expressions E_{tc} that affect the trip count using the SSA numbering [4, 88] of boolean variables in the loop termination condition to find definitions of these variables within the loop, as described in Secs. 6.1.2 and 6.2.1.

2. Repeatedly perform symbolic execution on the loop body, merging after each iteration and considering only pointer and boolean locations, until a fixed point approximating the values of all pointer and boolean locations at the loop header is reached. This stage, which completely ignores integer operations, ensures that pointer values are stable within subsequent iterations over the loop body.

3. Compute the dynamic field relation graph given the pointer values in the loop header and intersect the equivalence classes generated from (in)equality relationships in the dynamic FRG with the integer relationships guaranteed to hold on entry to the loop, as described in Sec. 6.3.

4. Replace the values of all integer locations with fresh symbolic constants according to the equivalence classes, such that all integer locations within any given equivalence class are given the same symbolic constant that is distinct from the symbolic constant for any other equivalence class.

5. Record the initial values of each integer location in a table I_l, mapping from the integer location to the set of values the integer location attains during successive iterations.

6. Symbolically execute the loop body. As expressions in the set E_{tc} are evaluated, record their values in the table I_e, a mapping from the expression to the set of values the expression attains.

7. For each equivalence class, verify that the values at each integer location within the equivalence class are equivalent following the loop's execution; if they are not, split the equivalence classes to ensure that equivalence is retained in all classes. If any equivalence class has been split, clear the I_l and I_e tables and return to step 4 given these new equivalence classes.

8. Record the values of each integer location in the table I_l.

9. Symbolically execute the loop body $m+1$ more times (where m is the maximum degree induction variable that is recognizable), recording expression values in the table I_e and integer location values in I_l at each evaluation and after each iteration, respectively.

10. Perform symbolic differencing [46] (described in Sec. 6.1.1) to compute induction expressions for the locations in I_l and the expressions in I_e with respect to a fresh iteration variable i.

11. Compute the trip count using the algorithm presented in Sec. 6.2.1.

Following this sequence of steps, the effects of the loop have been memoized for the initial conditions. The loop may now be symbolically executed given some set of initial conditions (that must be approximated by the initial conditions provided in the loop analysis phases), to perform static checking or to determine the results of the loop's execution. For the former case, we perform the following steps to simulate the execution of all iterations of the loop simultaneously:

1. Replace the value at each location with its induction expression, if it is inductive. If it is not inductive, replace the value with a fresh symbolic constant associated with the value's equivalence class.

2. Substitute the values of each integer location in the loop preheader for the symbolic constants used to initialize the corresponding location in the generalized loop analysis throughout the program state and in the trip count tc.

3. Let the iteration variable i fall in the range $[1 : tc]$.

4. Symbolically execute the loop body.

It is often required that one determine the side effects of a loop's execution, for instance when symbolically executing the outer loop of a loop nest while building the tables I_e and I_l. The procedure is equivalent to that of simulating all loop iterations, but instead of letting i fall in the range $[1 : tc]$, we let $i = tc$. We therefore simulate the last loop iteration only, to determine the state of the program after the loop has completed its execution.

6.5 Summary

Higher-level loop analysis presents many challenges not addressed by traditional loop analysis techniques, due to the extra levels of abstraction present in the program representation. Coupled with a precise integer analysis such as lazy symbolic range propagation within a symbolic execution framework, symbolic differencing is able to detect many forms of induction variables that elude traditional analyses. Symbolic differencing is also applied to loop trip count estimation for nontrivial loops.

Some important integral values are not inductive, requiring a loop analysis to be overly conservative regarding their values. Using the dynamic field relation graph introduced in Sec. 5.5, we can retain equivalences among integer values even when the values themselves cannot be precisely determined.

The loop analysis techniques described in this chapter are sufficient to analyze complicated loops involving monotonic variables, complex loop termination conditions, and induction variables modified via pointers in subroutines. The real-world need for such an analysis is illustrated by the loop in Fig. 3.9.

CHAPTER 7
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 [5] specifies the requirements of many STL algorithms in terms of higher-level semantic properties, in the manner shown in Fig. 1.1. We do not attempt to constructively prove that algorithms introduce certain properties, but instead rely on hand-written algorithm specifications that assert the appropriate semantic properties. The specifications described in this thesis and implemented in STLlint provide a method of "tagging" objects with other objects of varying types and accessing those "tag" objects later in the program via multiple dynamic classification. 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. 7.1 illustrates an example where the author has omitted the proper sorting predicate, 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);
```

Even by limiting ourselves to semantic properties useful within the C++ STL, there are a large number of combinations that must be considered. The STL contains more than 70 algorithms and 8 different container types, with many interesting interactions among them. We must therefore consider the incremental cost of introducing checking for a new semantic property. For instance, to add proper checking

```
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 7.1: 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, because it relies on the default < ordering.

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* [80, 97] that describe the behavior of algorithms and allow us to reduce the cost of introducing new semantic checks. We describe algorithm concepts in Sec. 7.1, describe their use in Sec. 7.2, and briefly sketch the implementation in Sec. 7.3..

7.1 Algorithm concepts

For checking programs using the STL, 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 verify that the incoming sequence is sorted; the Sorting concept assert that the sequence is sorted; and the Write concept assert that the sequence is no longer sorted, because values have been rearranged or overwritten. 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 former. The Semple language supports the introduction, removal, and verification of semantic property tags (such as "sorted") via multiple dynamic classification 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 details of the decision process that supports these semantics require that the refinement relationships between algorithm concepts be considered, to allow customizations at the least specific concept where a particular semantics should apply.

Algorithm concepts are arranged within a concept lattice [120], as is common with other (data type) concepts. Fig. 7.2 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.

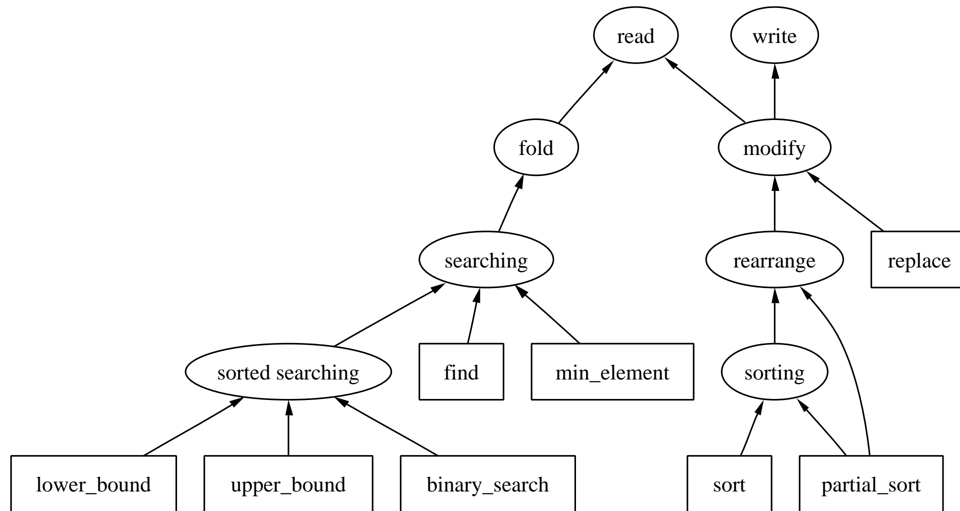


Figure 7.2: Partial algorithm concept lattice describing Standard Template Library algorithms

To determine the effects of an algorithm, we must find 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. The algorithm concept lattice remains static irrespective of the current axis, but each traversal halts at an algorithm concept customized for that axis. For instance, the search for a customization point for the `replace` algorithm along the sortedness axis would find the WRITE concept; thus, replacing elements in a sequence results in the sequence that is not necessarily sorted. 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, reducing

the effort required to introduce extensions to a generic library specification. Introducing a new algorithm requires only that one state the concepts that the algorithm models, after which the new algorithm 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 revisiting every algorithm for every new semantic check, and provides the additional benefit of being modular: STLlint allows one to toggle the checking of various semantic properties, such as "sortedness" and "removedness", from the command-line interface, allowing one to decide between scalability and specific semantic checks.

7.2 Customization via events

The customization of an algorithm concept for a particular axis involves injecting additional specification code at defined points in the algorithm's execution. We refer to these points as *events* and permit customization for any event of any algorithm concept. STLlint defines two types of events. The first, an *entry* event, allows customization code to be executed prior to the actual algorithm, and is typically used to perform verification of preconditions, e.g., the sequence must be sorted. The second, an *exit* event, executes just before control is returned to the caller of the algorithm, and is typically used to assert postconditions, e.g., the sequence forms a heap. The process of selecting customization code to execute for an algorithm is akin to the *weaving* process of Aspect-Oriented Programming [59, 60], where the entry and exit events correspond directly to *before* and *after* advice [59].

Throughout this thesis, we have presented higher-level static analysis for the Semple language, implicitly assuming that specifications could be written in another language and transformed into Semple with little effort. However, an extensible implementation of algorithm concepts and events must permit designers of active libraries [112, 114] to introduce new algorithm concepts, events, axes, and customizations in a natural way, which implies support in the source language.

```
struct read_sequence {};
struct write_sequence {};
struct fold : refines<read_sequence(_1, _2)> {};
struct modify_sequence : refines<read_sequence(_1, _2),
                                 write_sequence(_1, _2)> {};
struct rearrange_sequence : refines<modify_sequence(_1, _2)> {};
struct sorting : refines<rearrange_sequence(_1, _2)> {};
struct sort : refines<sorting(_1, _2, _3)> {};
struct partial_sort : refines<sorting(_1, _2, _4),
                              rearrange_sequence(_2, _3)> {};
struct searching : refines<fold(_1, _2)> {};
struct sorted_searching : refines<searching(_1, _2)> {};
struct lower_bound : refines<sorted_searching(_1, _2, _4)> {};
struct upper_bound : refines<sorted_searching(_1, _2, _4)> {};
struct equal_range : refines<sorted_searching(_1, _2, _4)> {};
```

Figure 7.3: C++ representation of the algorithm concept lattice in Fig. 7.2

STLlint's algorithm concepts and events are therefore implemented entirely within C++, using techniques familiar to C++ library designers. Aside from simplifying the process of introducing extensions for users, this also simplifies the static analysis engine, which need not consider algorithm concepts in its execution.

In STLlint, algorithm concepts are represented by otherwise empty C++ class types, with concept refinement represented by inheritance [5, 93]. Similarly, axes are represented by C++ class types and a set of C++ functions that operate on certain algorithm concepts. Fig. 7.3 illustrates the C++ representation of algorithm concepts.

Each algorithm and algorithm concept accept parameters describing the values on which they operate. The `partial_sort` algorithm [5, §25.3.1.3], for instance, accepts four parameters: `first`, `middle`, `last`, and `comp`; the corresponding algorithm concept accepts these same four parameters. On the other hand, the Sorting concept describes its behavior in terms of the sequence it is sorting and the ordering predicate, and therefore accepts three parameters: `first`, `last`, and `comp`. Thus when the PartialSort concept describes its refinement relationship with the Sorting concept, it must provide a mapping from its parameters to the parameters oo the Sorting stating that the sequence [`first`, `middle`) will be sorted using the relation `comp`. This information is encoded with a function type [5, §8.3.5] that provides the arguments to the refined concept in terms of the positions of the arguments

```
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 7.4: Skeletal implementation of the `lower_bound` algorithm including user annotations associating the implementation with the LOWER_BOUND concept

in the concept being described. For instance, PARTIALSORT concept describes its refinement of the SORTING concept as `sorting(_1, _2, _4)`, meaning that the first, second, and fourth arguments to PARTIALSORT (`first`, `middle`, and `comp`) are to become the arguments of SORTING, in that order. Similarly, the PARTIALSORT algorithm concept also rearranges elements in the sequence [`first`, `last`) by refining REARRANGE via `rearrange_sequence(_2, _3)`.

Each algorithm must explicitly state the algorithm concept it models, shown in Fig. 7.3 and Fig. 7.4, providing information about its arguments and return value. This information is provided to STLlint's event mechanism, which 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. 7.4 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 customiza-

tion (e.g., "sortedness"), and the function parameters provided by the algorithm to the event. Fig. 7.5 illustrates the `on_entry` handler responsible for uncovering the inconsistent predicate in Fig. 7.1. 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 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 postconditions effectively via algorithm concepts. However, this decoupling introduces additional levels of abstraction, which implies that semantic checks are placed 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 at the point of invocation.

7.3 Implementation of static event dispatch

The event dispatch mechanism, which matches an event (such as the `lower_bound` event in Fig. 7.4) with the appropriate event handlers (such as the sorted searching precondition handling in Fig. 7.5), executes at compile time to produce static bindings for the underlying static analysis engine. The generation of static bindings (i.e., direct function calls) ensures that the resulting program representation seamlessly merges specifications with all of the necessary entry and exit handlers without requiring static analysis of dynamic invocations.

Events are declared as in Fig. 7.4, where the type of each event object contains:

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

Figure 7.5: Implementation of the `on_entry` event handler 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`.

- The algorithm concept that the algorithm models, e.g., `lower_bound`.
- The signature of the algorithm, encoded as a function type. The function type `ForwardIterator(ForwardIterator, ForwardIterator, const T&, Compare)` describes an algorithm that accepts two forward iterators, a constant reference to a value of type `T`, and a comparison function object, and then returns a forward iterator.

The event object itself is constructed given the actual algorithm parameters. The event object's constructor then invokes the `on_entry` handlers. For algorithms with a non-`void` return type, the event object is called given the algorithm's result, which invokes the `on_exit` handlers and returns the (possibly modified) algorithm result. For algorithms with a `void` return type, the event object invokes `on_exit` handlers when it is destructed. The actual invocation of event handlers requires the detection of available handlers for that event and traversal of the algorithm concept lattice.

Event handlers are detected using argument-dependent lookup [5, §3.4.2] and C++ template metaprogramming techniques [22, 83, 111], including argument forwarding [53], metaprogramming control of the overloading mechanism [54, 110], and detection of valid expressions at compile time [64]. The result of the template metaprogram is either the best event handler for the given axis, algorithm concept,

and algorithm arguments; an error, if the selection of event handlers is ambiguous; or an indicator that no handler exists.

Static event dispatch traverses the algorithm concept lattice, which is represented indirectly via C++ inheritance as in Fig. 7.3. The dispatch mechanism employs the previously discussed event-handler detection metaprogram to determine if a handler exists at the most specific concept for an algorithm. If a handler does exist, that handler is executed and the process terminates. If no handler exists, the dispatch algorithm is recursively invoked for each of the superiors of the algorithm concept, after mapping each argument in the algorithm concept to the corresponding argument(s) of its superiors. For instance, consider the *exit* event of the `sort` algorithm along the "sortedness" axis. We begin our search at the SORT concept in Fig. 7.2. There is no handler for this particular event, therefore we follow the refinement edge to the nearest superior concept, SORTING. There exists an event handler for this concept, previously described, which states that the sequence is sorted according to the given ordering relation. The algorithm executes this event handler, then terminates for this event/axis combination (although other axes may then be considered).

7.4 Summary

The introduction of multiple dynamic classification into the Semple language permits the use of semantic tags, such as "sorted" or "removed", that may be attached to specifications, providing additional semantic information that can be verified by a static analysis. Through the use of algorithm concepts and template metaprogramming, these tags and the operations that manipulate them can be organized such that different semantic modules remain completely distinct, but interact smoothly. The overall effect is the production of a highly extensible but loosely coupled system that does not introduce a significant burden to the specification author.

CHAPTER 8
STLlint: an extensible static checker for the STL

The higher-level static analyses presented in this thesis are implemented in the extensible static checker STLlint. STLlint checks C++ programs for unsafe uses of the Standard Template Library, including iterator invalidation and movement, container/iterator interactions, and high-level semantic properties of algorithms. STLlint follows the model of the original `lint` [56] static checker for C, in that it retains a compiler-like command-line interface and performs checking of programs beyond what a compiler would do. Where STLlint detects potential errors in the program with respect to the use of the STL, it emits diagnostics describing the error and the location in user code that triggered the problem.

8.1 Scope

STLlint's semantic specifications cover the functions and data types presented in sections 21, 23, 24, 25, and 26.4 of the ANSI/ISO C++ standard [5], checking the vast majority of function preconditions and modeling all important data structures.

STLlint is designed to check C++ programs that make extensive use of the STL and other parts of the C++ standard library, including user extensions to the STL. While it does perform some checking at the language level, including the detection of NULL pointer dereference operations and memory leaks, other static checking systems with less expensive analyses may perform better for programs not using the higher-level abstractions of the STL.

STLlint includes specifications for the STL, but it is also extensible to other C++ software libraries. An active C++ library [112, 114] can provide specifications for its algorithms and data structures, using the features described in Chap. 7, thereby allowing STLlint to be used as a general C++ library-conformance checker.

```
09 vector<Student_info> extract_fails(vector<Student_info>& students)
10 {
11  vector<Student_info> fail;
12  vector<Student_info>::iterator iter = students.begin();
13  vector<Student_info>::iterator end_iter = students.end();
14  while (iter != end_iter) {
15    if (fgrade(*iter)) {
16      fail.push_back(*iter);
17      iter = students.erase(iter);
18    } else
19    ++iter;
20  }
21  return fail;
22 }
```

Figure 8.1: [`fails_iters_bad.cc`] **A misguided optimization of the example code in Fig. 3.9, which results in a comparison against the singular iterator** `end_iter`

8.2 Examples

We have tested STLlint on a variety of examples distilled from C++ and STL textbooks [8, 62, 72, 81]. The following are selected examples that showcase the unique capabilities of STLlint; the STLlint test suite is discussed in greater detail in Sec. 8.4.

8.2.1 Iterator invalidation

The example in Fig. 8.1 closely resembles the example in Fig. 3.9 on page 47. The crucial difference is that Fig. 8.1 contains a "misguided optimization" provided by the authors [62] to illustrate the effects of iterator invalidation. The iterator `end_iter` is invalidated by the call to `students.erase`, resulting in an invalid comparison within the loop conditional. Moreover, `iter` will run past the end of the `students` container, causing additional errors. STLlint correctly identifies the errors when given the complete sample program containing this error:

```
"fails_iters_bad.cc", line 15, warning: attempt to dereference a
      past-the-end iterator

                  if (fgrade(*iter)) {

in call to function operator* at "fails_iters_bad.cc", line 15
  in call to function extract_fails at "fails_iters_bad.cc", line 46

"fails_iters_bad.cc", line 16, warning: attempt to dereference a
      past-the-end iterator

                          fail.push_back(*iter);

in call to function operator* at "fails_iters_bad.cc", line 16
  in call to function extract_fails at "fails_iters_bad.cc", line 46

"fails_iters_bad.cc", line 19, warning: attempt to increment a
      past-the-end iterator

                          ++iter;

in call to function operator++ at "fails_iters_bad.cc", line 19
  in call to function extract_fails at "fails_iters_bad.cc", line 46

"fails_iters_bad.cc", line 46, warning: attempt to compare a singular
      iterator

          vector<Student_info> fails = extract_fails(vs);

in call to function extract_fails at "fails_iters_bad.cc", line 46

4 errors detected in the compilation of "fails_iters_bad.cc".
```

8.2.2 Writing to a past-the-end iterator

Fig. 8.2 provides a clear example of the difference between library-level and language-level semantics. For every known implementation of an STL `vector` that does not perform run-time checking, the example will compile and execute correctly because no *language* requirements are violated. However, the requirements of an STL `vector` are violated: the call to `reserve` does not, in fact, introduce any elements into the container. Thus writing any nonzero number of elements via the iterator `results.begin()` is inherently unsafe, as it is a past-the-end iterator. Additionally, the subscript operation within the `for` loop will receive out-of-bounds

```
15 std::vector<int> values; // fill values
16 std::vector<int> results;
17 results.reserve(argc);
18 std::transform(values.begin(), values.end(),
19                results.begin(),
20                transmogrify);
21
22 for (int i = 0; i < values.size(); ++i) {
23   std::cout << results[i] << ' ';
24 }
```

Figure 8.2: [`meyers_scary.cpp`] **An example adapted from Meyers [72] that executes predictably under unchecked STL implementations, but violates STL requirements**

indices, again violating STL requirements. A previous paper [42] details the style of `vector` implementation for which the example code is correct with respect to the C++ language.

STLlint's library-level static checking uncovers the problems in Fig. 8.2, producing diagnostics that reveal both problems. Fig. 8.11 provides an alternative, more graphical view of the same error messages and will be discussed in Sec. 8.5. STLlint produces the following four correct diagnostics:

```
"meyers_scary.cpp", lines 18-20, 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 "meyers_scary.cpp", lines 18-20

"/home/gregod/Projects/GPG/EDG/libcomo/stl_algo.h", line 516, warning:
      attempt to dereference a past-the-end iterator

      *__result = __opr(*__first);

in call to function operator* at "/.../libcomo/stl_algo.h", line 516
  in call to function transform at "meyers_scary.cpp", lines 18-20

"/home/gregod/Projects/GPG/EDG/libcomo/stl_algo.h", line 515, warning:
      attempt to increment a past-the-end iterator

    for ( ; __first != __last; ++__first, ++__result)

in call to function operator++ at "/.../libcomo/stl_algo.h", line 515
  in call to function transform at "meyers_scary.cpp", lines 18-20

"meyers_scary.cpp", line 23, warning: subscript may meet or exceed
      container size

      std::cout << results[i] << ' ';

in call to function operator[] at "meyers_scary.cpp", line 23

4 errors detected in the compilation of "meyers_scary.cpp".
```

There are two trivial ways in which the example code from Fig. 8.2 can be fixed. The first is to replace the call to `reserve` with a call to `resize`, which does change the size of the container by inserting elements. The second is to replace the use of the `results.begin()` iterator with a `back_insert_iterator` [5, §24.4.2] that transforms writes to insertions at the back of the container. If either solution is implemented, STLlint correctly verifies that the code in question contains no safety violations.

```
15 vector<int> v;
// fill v
20 sort(v.begin(), v.end(), less<int>());
// ...
24 vector<int>::iterator i = find(v.begin(), v.end(), 42);
```

Figure 8.3: [`sort_slow_search.cpp`] **Illustration of inefficient algorithm selection in a user program. STLlint will suggest an algorithmic optimization based on `lower_bound`.**

8.2.3 Inefficient algorithm selection

With over 70 algorithms, selecting the most appropriate STL algorithm to apply to any particular task can be daunting. Especially within a generic algorithm, it is often the case that particular assumptions on the input sequence may lead to more efficient implementations, but these assumptions cannot be made in general. However, if these assumptions could be validated, a different, more efficient algorithm could be selected. Fig. 8.3 illustrates one such instance, where the user of the container `v` has applied a linear search. In this instance, a binary search such as `lower_bound` would provide the same semantics but would operate in logarithmic time instead of linear time. STLlint's semantic descriptions use the "sortedness" information introduced by the `sort` algorithm to produce an optimization hint. Note that STLlint does not attempt to perform the transformation, which would require additional analysis, but it can call attention to potential optimizations:

```
"sort_slow_search.cpp", line 24, warning: potential optimization: the
      incoming sequence [first, last) is sorted, but will be searched
      linearly with this algorithm. Consider replacing this algorithm
      with one specialized for sorted sequences (e.g., lower_bound)

    vector<int>::iterator i = find(v.begin(), v.end(), 42);

in call to function find at "sort_slow_search.cpp", line 24

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

8.3 Design and implementation

STLlint combines the Semple high-level static analysis engine with the Edison C++ front end [26] and specifications for the Standard Template Library. Fig. 8.4

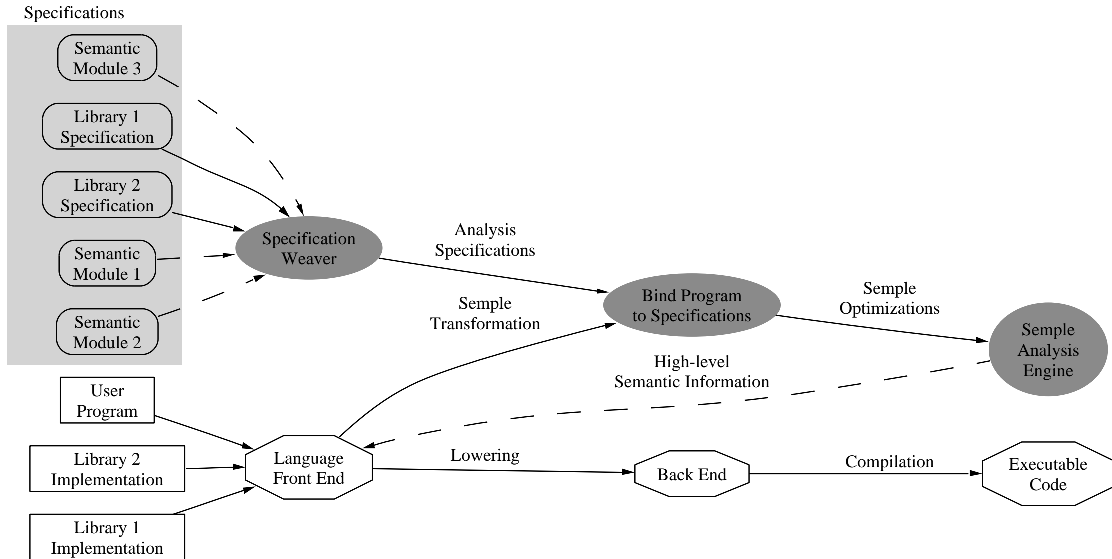


Figure 8.4: Overview of the STLlint static checker.

illustrates the flow of information amongst the major components of STLlint. Of particular importance are the following modules and transformations:

- Specifications: STLlint provides specifications for all STL containers, iterators, and algorithms, the formulation of which has been discussed in Sec. 3.3.1 and Sec. 7.2 and studied in-depth in Chap. 5 and Chap. 6. Sec. 8.3.2 describes the architecture and design of these specifications.

- Specification weaver: This component, which combines disjoint specifications and semantic analysis modules, is implemented as a C++ template metaprogram described in Sec. 7.3. The term "weaver" is applied here due to the similarity between this process and the weaving process of Aspect-Oriented Programming [60].

- Semple transformation: The transformation from C++ to Semple is routine given the semantics of both languages, and is similar to the transformation to C. Sec. 8.3.1 describes the transformation for Semple features with no direct C++ counterpart.

- Semple optimizations: Prior to the symbolic execution of the user program and library specifications, the Semple representation of the program is optimized

```
template<typename Classification, typename T>
  void classify(T& object,
                const Classification& classification = Classification());
template<typename Classification, typename T>
  void declassify(T& object);
template<typename Classification, typename T>
  Classification& as_a(T& t);
template<typename Classification, typename T>
  bool is_a(T&);
```

Figure 8.5: Multiple dynamic classification primitives provided by STLlint

for analysis via expression propagation (a generalization of constant and copy propagation), goal-directed inlining (Sec. 6.2.2), and elimination of multiple subroutine return statements. While some of these transformations (goal-directed inlining in particular) do not result in smaller or faster executable code, they can improve the precision or efficiency of the analysis.

- Semple analysis engine: The Semple analysis engine performs symbolic execution on the Semple representation of the user program and library specifications, using the techniques described in this thesis. Any diagnostics are passed to the C++ front end interface to be presented to the user.

8.3.1 Static checking primitives

STLlint provides several Semple primitives accessible via C++ constructs, including assertions and assumptions, operations to support multiple dynamic classification, and the `foreach` statement. All of the constructs are written in valid C++ but receive special treatment by the C++-to-Semple transformation, allowing one to write component specifications using the full expressive power of C++. C++ library designers can therefore write specifications in a familiar language, reusing library primitives that may be common both to the implementation and the specification. Writing specifications in C++ also permits a programmer to gradually transition from checking programs against the library implementation to checking programs against the library interface specification.

Assertions and assumptions are implemented as C++ functions. Each function

```
void _M_invalidate_from(size_type __n)
{
  STLLINT_FOR_EACH(iterator __i) {
    if (__i._M_sequence == this && __i._M_position >= __n)
      __i._M_version = 0;
  }
}
```

Figure 8.6: Definition of the `_M_invalidate_from` method, used by the STLlint `vector` specification, that invalidates all iterators referencing the current container whose positions equal or exceed the input value `__n`

accepts a boolean parameter (the condition that will be asserted/assumed) and the assertion function accepts a second argument containing a string literal. Invocations of these functions are translated directly to the Semple assertion and assumption statements.

Multiple dynamic classification in Semple involves the `classify` and `declassify` statements (Sec. 4.2.5), the `is-a` expression (Sec. 4.2.4.1) and the `as-a` expression (Sec. 4.2.4.4). Fig. 8.5 illustrates the declarations of the corresponding C++ functions. The template parameter `Classification` provides the dynamic classification type, and is generally specified explicitly, i.e., `is_a<sorted>(my_container)`.

The Semple `foreach` loop (Sec. 4.2.5) iterates over all values of a particular type in the abstract program state. This construct is represented by the `STLLINT_FOR_EACH` function-like macro,[2] which accepts a single parameter—the declaration of the variable that will iterate over each value—and is followed by the loop body. Fig. 8.6 illustrates the use of the `STLLINT_FOR_EACH` macro to invalidate iterators in a `vector`.

8.3.2 C++ standard library specifications

STLlint provides specifications of STL containers, iterators, and algorithms, described throughout this thesis. There are many similarities among the specifications of various containers and their iterators in particular, which can be coalesced

[2]Macros are generally considered crude, but in this case the macro itself expands to an appropriate `for` loop and therefore retains the basic lexical properties of loops that might otherwise be lost.

by employing a wrapper model of safe sequences and safe iterators. With this wrapper model, we drastically reduce the amount of specification code required, since one safe iterator wrapper suffices for all STL container iterators, and permit users to apply the wrappers to their own container and iterator types.

Safe iterators are typically applied in run-time–checked implementations of the STL [35, 38, 49], and are implemented as an adaptor over the underlying, unsafe container iterator. These adaptors (or wrappers) contain the extra member data and assertions required to perform run-time checking. STLlint's specifications employ the same idiom, with two differences: the specification method in STLlint relies on the Semple language and STLlint does not forward any operations to the underlying iterator because they are unnecessary. The resulting architecture mimics that of various "safe" STL implementations, while providing a simple method of reusing iterator specifications. Appendix A includes the definition of the `_Safe_iterator` class template, which serves as the basis for all STLlint iterator specifications.

Safe sequences are analogous to safe iterators for containers, and are employed as run-time–checked adaptors in some safe STL implementations [35, 38]. The iterators exposed by safe sequences are safe iterators, which communicate with the sequence via a common base class template for all safe sequences that keeps a list of all iterators referencing it. Each safe container inherits the behavior of the underlying unsafe container implementation and the base class template, introducing additional checking and forwarding operations to the underlying unsafe container. As with safe iterators, STLlint's specification employs the same model, but eliminates forwarding to the underlying implementation and simplifies the specification. Appendix B provides a specification of the STL `vector`.

Algorithm specifications, as shown in Fig. 7.4, contain event objects that allow customization of the checking performed for STL algorithms. With many simple algorithms, STLlint specifications need only introduce this event object—which associates the implementation with its algorithm concept—because the algorithm implementation is a suitable specification for mimicking its high-level postconditions. For more complex algorithms that require time-consuming analysis or would produce imprecise analysis results, STLlint provides an alternative algorithm speci-

fication that verifies preconditions and asserts postconditions but does not simulate the algorithm's execution.

The close architectural ties between STLlint's specifications and the run-time adaptors employed by safe STL implementations hint at the ability to combine both into an efficient, safe STL implementation. This topic is considered in more detail in Sec. 10.1.

8.3.3 Implementation

The implementation of STLlint follows closely the architecture presented in Fig. 8.4, separated into three distinct modules: the Semple static analysis engine, the STL specifications, and the C++-to-Semple transformation.

The Semple static analysis engine, including Semple optimizations and tools to aid in processing of programs written in the Semple language, consists of approximately 15k lines of C++ source code. Our implementation of lazy range propagation is based on the GiNaC symbolic computation framework [10], which we have extended to support symbolic variable bounds. While we have only experimented with C++, the Semple language and static analysis are language-independent.

The high-level specifications for the C++ Standard Template Library comprise roughly 6k lines of C++ code. They describe the semantic behavior of all containers, iterators, and algorithms within the STL and mimic precisely the interfaces of the STL components they describe. These specifications are further divided into smaller semantic modules that perform checking for certain behaviors. For instance, the module that checks heap properties is 136 lines of C++ code, whereas the module that checks "sortedness" properties is 135 lines of C++ code.

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 [26] to the Semple intermediate representation. This 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.

8.4 Experimental results

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, some of which have been written specifically to test various capabilities of STLlint and others that have been collected from several sources, including:

- *Accelerated C++* [62]: This book teaches introductory C++ using STL and avoids lower-level constructs such as explicit memory management, arrays, and type casting. 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* [72]: 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* [38]: 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.

8.4.1 Precision

STLlint performs well on the constructed test suite, achieving a very low false positive rate of 0.59%. The false positives come from several sources, but of particular interest are the observations that several examples require precise modeling of the values stored in a container and container iterators that are stable, such as `list`

```
map<int, int> v;
for (int i = 0; i < 20; ++i) {
  v[i] = 20-i;
}

map<int, int>::iterator at = v.begin();
advance(at, 3);
```

Figure 8.7: STLlint is unable to deduce that the values of `i` across all iterations are distinct, and therefore conservatively determines that the resulting size of the container `v` varies falls in the range $[0 : 20]$.

and `map`. Stable iterators are iterators that are invalidated only when the elements they refer to are removed.

Fig. 8.7 provides an example where STLlint's inability to precisely model the values stored within an STL `map` results in false positives within the test suite. Within the loop, each of the values of `i` are distinct. Therefore, the implicit insertion performed by the bracket operator will occur during each iteration. However, STLlint is unable to verify that the value of `i` is disjoint from any values previously inserted into the `map`, and therefore conservatively determines that the `map` size falls between zero and twenty. Subsequent operations, such as the `advance` call, therefore trigger false positives. The test case containing this code fragment, adapted from the libstdc++ test suite [38], accounts for 40% of the false positives within the test suite. It is also worth noting that the introduction of the assumption `_STLlint::assumption(v.size() == 20);` following the loop eliminates all false positives from this test case.

Fig. 8.8 illustrates one instance where verification of the correctness of the second loop requires that the end iterator's position remain equivalent to the size of the container. This fact is lost due to the (conditional) call to `insert`, which must selectively shift the positions of iterators to accommodate the insertion. For instance, if we insert a new value at position p, all iterators whose position is $\geq p$ will be incremented one step. During the analysis of the loop, the position of the iterator `it` cannot be bounded and it is therefore unknown whether it precedes the `end` iterator, so the insertion may move the `end` forward one step or may leave it in

```
bool
operator<(std::pair<int, int> const& lhs, std::pair<int, int> const& rhs)
{ return lhs.first < rhs.first; }

// ...

typedef std::multiset<std::pair<int, int> >::iterator iterator;
std::pair<int, int> p(69, 0);
std::multiset<std::pair<int, int> > s;
for (p.second = 0; p.second < 5; ++p.second)
  s.insert(p);
for (iterator it = s.begin(); it != s.end(); ++it)
  if (it->second < 5)
    {
      s.insert(it, p);
      ++p.second;
    }
```

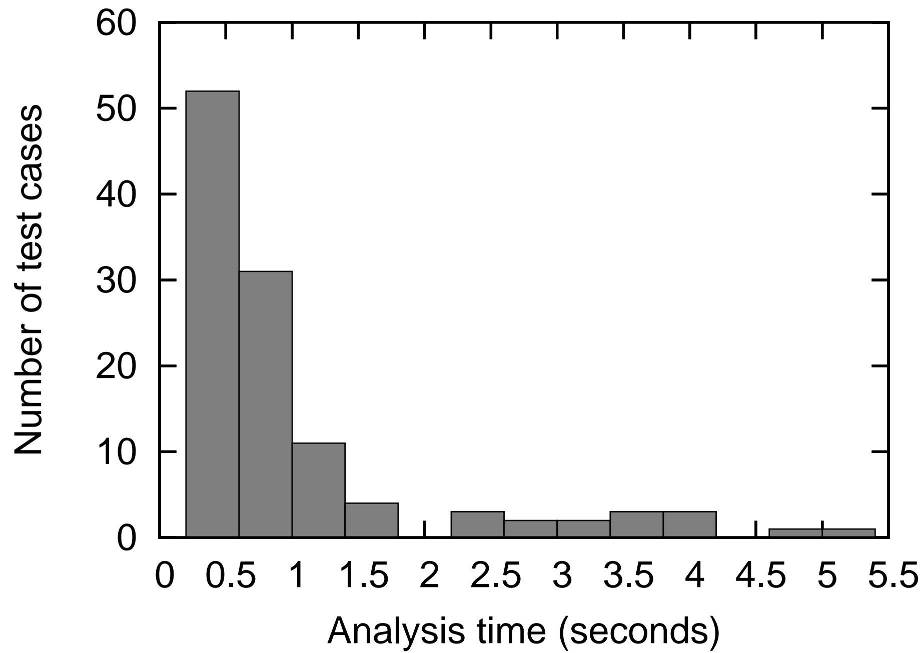
Figure 8.8: STLlint is unable to verify the correctness of the second loop due to the insufficient modeling of the stable `multiset` iterators.

the same position. It is due to the latter case (which does not correspond to any real execution of the program) that we receive two false positives from this example.

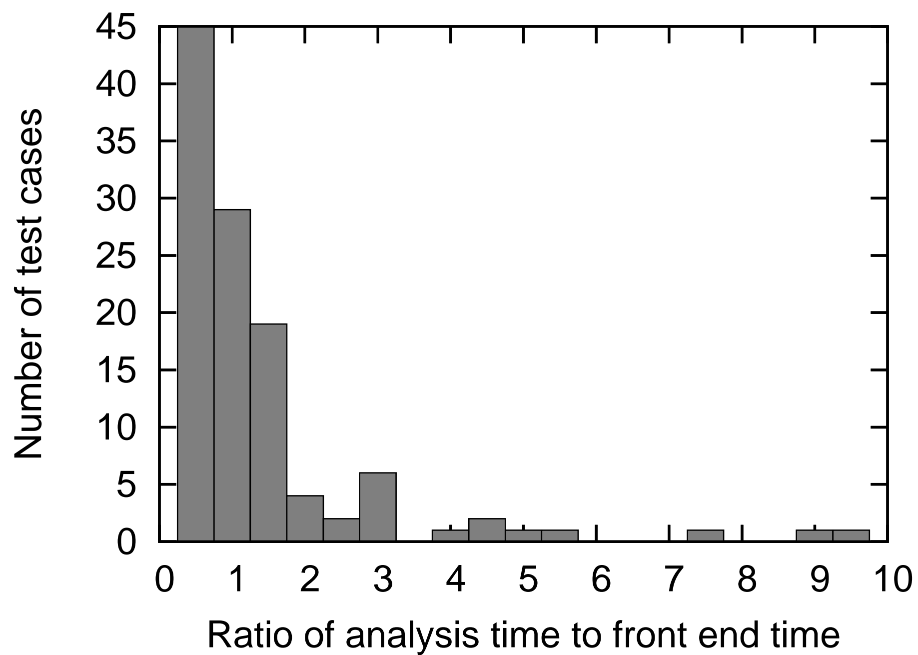
8.4.2 Efficiency

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. STLint itself employs version 3.0.1 of the Edison C++ front end [26] and was compiled with version 3.3.2 of the GNU C++ compiler [38] with a high optimization level (`-O3`).

The histogram in Fig. 8.9(a) illustrates the analysis time required by examples in our test suite; most of the examples in our test suite are analyzed in under a second. To scale these results by program size, Fig. 8.9(b) provides a histogram of the ratios of analysis time to the time spent in the front end. While the majority of



(a)



(b)

Figure 8.9: 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

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. Table C.1 provides execution and front end times for each test case.

Table C.2 provides static program size statistics for the STLlint test suite, including the maximum number of fields in any class and the maximum access path length. The variation in analysis times does not correspond to the size of the combined Semple representation (including STL specifications), based on number of Semple statements, number of functions, or number of classes. However, of particular interest is the relatively small and consistent access path length, which affects the complexity of the construction of the dynamic field relation graph (see Section 5.6). Across the entire stest suite, no access path has more than 5 steps, and we can expect this value to stay reasonably constant because very long access paths indicate strong coupling among many different objects, and would likely contribute to unmaintainable code.

Inspection of the most expensive tests reveals that a significant amount of time is spent within the integer comparison algorithm. Fig. 8.10 illustrates the relationships between analysis time and the number of integer comparisons and variable replacements that have been performed. There is a strong linear dependency in both cases, indicating that analysis time increases with an increase in the number of integer comparisons and variable replacements. Table C.3 provides the number of comparisons and replacements performed by each test case.

The number of comparisons and replacements is not directly correlated with program size. However, there are certain operations in the test cases that introduce a large number of additional comparisons and/or replacements. For instance, the specification of the `insert` routine of an STL `map` or `set` employs the Semple `foreach` construct to conditionally increment the position of every iterator, introducing a new symbol $\alpha_i \in [0 : 1]$ for each iterator in the abstract program state. The effect of this operation is most obvious when comparing the test cases `map/invalidation/2.cpp` and `map/invalidation/2_unrolled.cpp`. The former was imported from the libstdc++ test suite [38] and includes the loop in Fig. 8.7; the latter is the same test case with the loop unrolled by hand. Each iteration

through the loop involves insertion into the `map`, including execution of the `foreach` loop and the introduction of many new symbolic variables. However, while the former test case executes the loop seven times—five to analyze the loop and two to determine its correctness and side effects—the latter test case executes the body of the loop twenty times. Table C.3 shows that the former test case, which contains the actual loop, requires only 10,000 comparisons whereas the latter test case, which contains the unrolled loop, requires over 3 times as many comparisons (35,000) and 2.5 times as many variable replacements. We therefore conclude that operations involving `foreach` loops that conditionally modify integer values, such as insertion into a `map` or `set`, introduce a large number of additional comparisons into the program that increase analysis time.

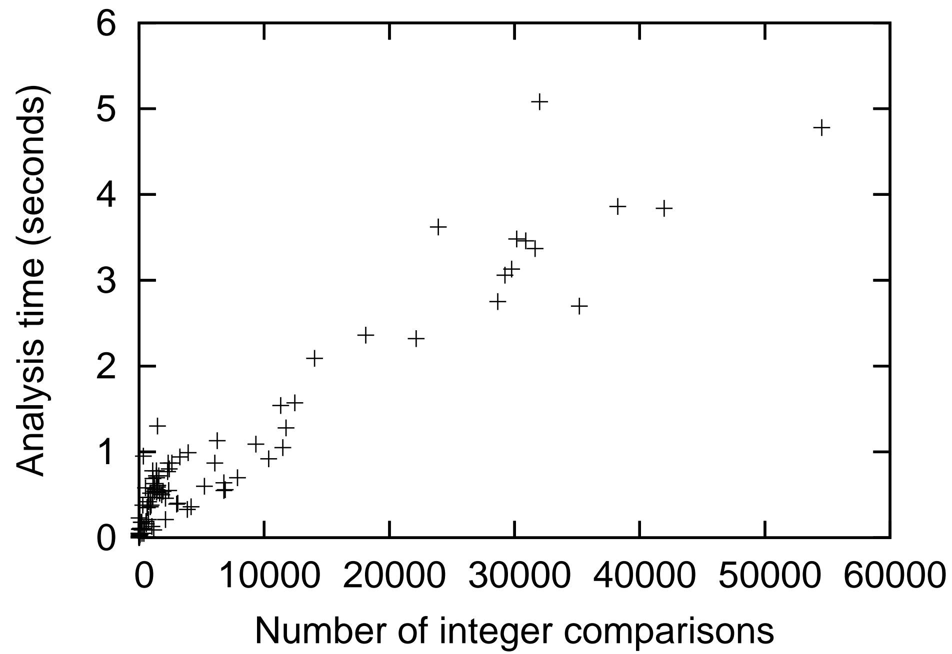
Overall, STLlint is reasonably efficient for small programs not employing operations that make heavy use of the Semple `foreach` construct, such as those that perform operations primarily on STL `vector`s, `deque`s, or the C++ standard library's `string` type. However, scalability is poor in the presence of data structures such as `map` and `set`. Sec. 10.3 discusses the possibility of applying other iteration models to the stable (`multi`)`set` and (`multi`)`map` iterators, which may potentially improve both analysis precision—see Fig. 8.8—and efficiency. However, the position-based model is both suitably precise and efficient for random access iterators as used in the STL `vector` and `deque` data structures.

8.5 Additional capabilities

This section details additional capabilities and features of STLlint that are nowhere else described but affect the user experience, including library-centric diagnostics and verification of generic algorithms.

8.5.1 Library-centric diagnostics

Sec. 7.2 alludes to STLlint's ability to associate errors found in event handlers deep within the call stack with the top-level library routine the user has invoked. Additionally, STLlint uses call stack information to group several symptoms of a problem into a single category and can associate particular library routines



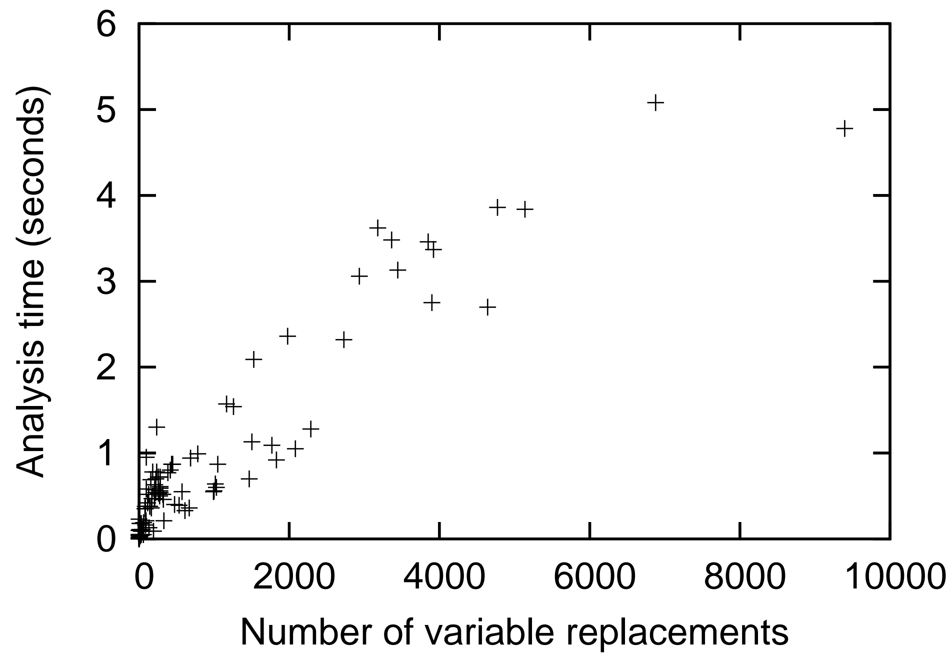


Figure 8.10: Plots illustrating the relationships between analysis time and the number of integer comparisons or variable replacements performed by lazy range propagation

"meyers_scary.cpp", lines 18-20, 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 "meyers_scary.cpp", lines 18-20

Implementation-specific symptoms:

- warning at "/home/gregod/Projects/GPG/EDG/libcomo/stl_algo.h", line 606
- warning at "/home/gregod/Projects/GPG/EDG/libcomo/stl_algo.h", line 605

```
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

Figure 8.11: Diagnostic output produced by STLlint that illustrates error message grouping and linking to external documentation

with library-provided documentation. Both of these capabilities are illustrated in Fig. 8.11, a partial screen capture displaying STLlint's diagnostics for the example in Fig. 8.2.

Error message grouping is conceptually accomplished by placing all diagnostics at the vertices within the invocation graph [27] where the diagnostics will be reported. Any diagnostic that occurs within the subtree of a node containing a diagnostic is considered a child (implementation-specific symptom) of that ancestor node. The effect is to suppress implementation- and specification-specific symptoms at the top level, but provide them to allow the user to explore the effects of, e.g., ignoring a particular precondition. In the example, the two child diagnostics refer to attempts to increment and dereference past-the-end iterators.

The bottom half of the diagnostic presentation in Fig. 8.11 is a reference to documentation regarding the library algorithm [8, 98], in this case the `transform` algorithm [5, §25.2.3]. STLlint permits documentation markup within its specifi-

cations containing links to external documentation sources. Whenever a documentation link is provided for an algorithm that has been used incorrectly, STLlint displays that documentation with the diagnostic.

8.5.2 Generic algorithm safety checking

Verification of generic algorithms has been explicitly excluded from the discussion of this thesis. However, we have experimented with the use of STLlint as a verifier for the safety of generic algorithms, in particular to verify that a generic algorithm does not violate the semantics of the concepts its parameters must model. For instance, within the STL, the differences between the InputIterator and ForwardIterator concepts [5, §24.1] are purely semantic, and we would like to verify that an algorithm declared to accept InputIterators will be correct for all models of an InputIterator.

The verification that a generic algorithm uses its parameters in a manner that is *syntactically* correct and type-safe has been solved in C++ by the use of archetypes [101, 119]. Archetypes are C++ constructs that mimic the least specific syntactic and type guarantees. If an algorithm represented by a function template can be instantiated with the archetypes corresponding to the concepts its parameters should model, the algorithm is syntactically correct.

STLlint extends the notion of syntactic archetypes to include the least specific semantics of any model of a particular concept. Our experiment consisted of constructing a syntactic InputIterator archetype and imbuing it with semantics that reflect the capabilities of input iterators. We then constructed small programs that invoked STL algorithms such as `adjacent_find` and `max_element` (both require ForwardIterators) given instances of the semantic InputIterator archetype. In these cases, STLlint produces diagnostics illustrating that the iterators have been used improperly. Verification of the same algorithms with a semantic ForwardIterator archetype yields no errors, because the algorithms restrict themselves to the behavior guaranteed by ForwardIterators. Thus, STLlint can be applied to the verification of generic algorithms via the use of semantic archetypes.

8.6 Limitations

The primary limitation of STLlint is that it does not attempt to support the entirety of the C++ language. STLlint supports only the subset of C++ that can be meaningfully transformed into Semple, assuming that the program is type safe and well-structured and ignoring certain C++ constructs, such as:

- Operations and data structures that break the type safety in the language, in particular unions and `reinterpret_cast`.
- Operations that can result in irreducible control-flow graphs, such as `goto`, `setjmp`, and `longjmp`.

The impact of this decision is expected to be minimal, because the C++ programs that conform to the high-level "STL style" of programming are typically very different from C++ programs that require low-level, unsafe operations or unstructured control flow. In our experience, only highly optimized generic libraries require these low-level facilities, and these libraries are better replaced by specifications as we have done with the STL.

Aside from Semple language limitations, STLlint imposes several limitations on its use that affect its applicability, including the lack of separate compilation, its inability to operate on program fragments, and its lack of contextual information (such as a backtrace showing the execution path along which the error occurs) in diagnostics. The impact of these limitations on users of STLlint bears further explanation.

The lack of separate compilation introduces the need for programs to be coalesced into a single, monolithic entity for analysis. While such a problem is trivially solved by creating a program database, the analysis of large programs as a single entity can be very expensive. This problem has not been evident with STLlint due to the limited size of the test suite. Should it become a problem for a particular program, STLlint's modular specifications permit one to easily disable checking of certain semantic properties (e.g., the "sortedness" property) to improve performance.

To counter scalability problems with program analysis, whole program analyses can be converted to fragment analyses [89, 90] that analyze various pieces of the program separately. However, STLlint does not support such fragment analysis, due to the complexity of constructing an initial state (in particular, aliasing relationships) for symbolic execution on a program fragment.

Finally, diagnostics emitted by STLlint provide only the source location that triggered the error, a static text string, and the call stack that lead to the error. From this information it can be hard to determine the cause of the error. For instance, isolating the statement that caused the invalidation of a particular iterator can be particularly vexing, and STLlint provides little assistance because any information about previous state (that could be useful for determining the cause) has been lost. This loss is an artifact of symbolic execution, which only retains state when required, e.g., when analyzing loops or parallel branches.

8.7 Summary

STLlint implements the higher-level static analyses presented in this thesis in an extensible static checker for programs using the STL. It is able to detect errors due to iterator invalidation, improper iterator movement, unsafe container/iterator interactions, and even high-level semantic properties regarding the ordering of sequences denoted by iterators. Within the context of our test suite, constructed from well-known C++ texts and test suites, STLlint performs well, with a low false positive rate of 0.59%.

The two most important problems with STLlint are scalability and precision. Scalability is the larger problem, because it is a fundamental limitation of context-sensitive, whole-program analyses. Precision, on the other hand, can be further improved by refining the specifications, in particular those of containers such as `map` and `set` whose iterators are relatively stable. Overall, the precision of STLlint's analysis and its ability to integrate well with active, generic C++ libraries make it useful as a tool for exploring and understanding particular software libraries.

CHAPTER 9
Conclusions

Higher-level static analysis for generic software libraries is markedly different from static analysis of lower-level constructs due to the greater use of abstractions. Many forms of abstractions, such as subroutines and classes, are invaluable to the programmer but hamper static analysis methods, resulting in poor precision, or are completely ignored. Higher-level static analysis addresses these problems by exploiting knowledge of interobject relationships and library interface specifications, using symbolic analyses that can effectively cope with language-level abstractions.

The abstractions present in software libraries, such as the rigid separation between library components and user programs via a well-defined interface, provide opportunities to simplify the static analysis task. By way of software specifications, we can reduce complex implementation behaviors to simple models, also gaining the benefits of library implementation independence. Moreover, object-oriented abstractions can be exploited to improve the precision of analysis further for a very important class of iterator movement problems. Other abstractions, such as subroutine abstractions and pointers, can be effectively handled with symbolic analysis methods, even in program loops with nontraditional induction variables.

Static checking for generic software libraries requires extensibility beyond what is required for more conventional software libraries. This extensibility can be effectively managed via algorithm concepts and semantic checks that are specified in terms of concepts instead of concrete algorithms.

STLlint is an extensible, library-level static checker for programs using the C++ Standard Template Library. It applies the techniques presented in this thesis and is capable of detecting many types of errors in the use of STL while retaining a low false positive rate. Like the STL interface, STLlint's specifications are extensible by any active library, and extensions integrate seamlessly with the STLlint system.

CHAPTER 10
Future work

As with any static analysis, the techniques presented here can—and should—be tested on a larger variety of software to provide a greater understanding of the patterns important for real programs. Such information is fed back into the development process to optimize an analysis both for improved precision and efficiency. With our extensible analysis, we can apply the same optimization strategy to the specifications of STL components, which greatly impact the performance of the entire system. The specifications of `map` and `set`, for instance, introduce a large number of additional symbolic integer values and have degraded STLlint's performance on certain programs.

However, there are three avenues for future work that are of particular interest, including integration of a higher-level static checker with run-time checking, construction of a fragment analysis, and researching other representations of higher-level iteration models.

10.1 Integration of static and run-time checking

Static checking and optimization are very often linked within compilers for safe programming languages. Array bounds checks, for instance, are performed in programming languages such as Java [7] but are optimized away whenever a static checker can determine that a particular subscripting operation will always be safe. Future research may consider the extension of this relationship to software libraries, where the library can provide additional checking that can be optimized away by a higher-level static analysis.

We have taken preliminary steps in this direction by mimicking the architecture of the GNU libstdc++ debug mode [38], which adds run-time checking of some STL preconditions, within the specifications of STLlint and have studied optimization for generic software libraries [43, 44, 95, 94, 96]. However, all of the optimizations studied have one important property: the difference between a "safe"

runtime version and an unsafe one has been purely a matter of additional program code. Run-time checking systems for the STL do not possess this property, because the implementations of safe components differ drastically from those of unsafe components. Therefore, providing the symbiotic relationship between safe libraries and static analysis may require in-depth knowledge of the design of the debugging components, and optimization based on this knowledge may require altering the types of program variables and propagating these changes throughout large sections of the program code.

10.2 Fragment analysis

Our experience with both run-time and static checking of STL usage errors has shown that many errors are localized within a small group of subroutines in a single source file. For these errors, the expense of whole program analysis is disproportionate to the actual amount of program code that needs analysis. For large programs, whole program analysis with the precision we require is not feasible.

Flow-insensitive program analyses can be adapted to fragment analyses by synthesizing a suitable driver program [89, 90] that includes all statements that may affect the program. With flow-sensitive analyses, the problem becomes harder because we must synthesize more types of program statements. In particular, we must determine the aliasing relationships that may occur in the invocation of a subroutine. These aliasing relationships are often obvious within the context of whole program analysis, but are hard to discern from the program source code. Similarly, we must consider all potential classifications for each object and the effect that the program fragment may have on classifications. Further research in the area of fragment analysis may be applied to improve the suitability of STLlint and our higher-level analyses to larger, more complex programs.

10.3 Other iteration models

In this thesis we have studied a model of iterators involving a reference to a sequence and an integral position. While this model is effective for containers providing random access iterators, such as `vector` and `deque`, it becomes cumbersome

for containers with stable iterators, such as `list` and `map`, because insertions into these containers involve shifting the positions of other iterations in the container. Future work may seek to uncover more suitable abstractions for these stable iterators, perhaps considering a model based primarily on iterator aliasing (such that one can invalidate iterators that may alias an iterator whose element is being removed) or on sets of valid range relationships (that need not contain distance information).

REFERENCES

[1] A. Aggarwal and K. H. Randall. Related field analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 214–220, June 2001.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[4] 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.

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

[6] S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specification. *Transactions on Software Engineering*, 26(1):55–69, January 2000.

[7] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1998.

[8] M. H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1999.

[9] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994.

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

[11] B. Bloom, I. Simmonds, D. Reimer, and M. Wegman. FERRET: Programming language support for multiple dynamic classification. Technical report, IBM T.J. Watson Research Center, 2001.

[12] W. Blume and R. Eigenmann. Symbolic range propagation. In *Proceedings of the International Parallel Processing Symposium*, pages 357–363, April 1995.

[13] R. Bodik and S. Anik. Path-sensitive value-flow analysis. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 237–251, 1998.

[14] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 321–333, 2000.

[15] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. *Lecture Notes in Computer Science*, 735:128–142, 1993.

[16] The Canvas project. `http://www.research.ibm.com/menage/canvas/`, 2001.

[17] P. Chan, D. Kramer, and R. Lee. *The Java Class Libraries: Supplement for the Java 2 Platform*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1999.

[18] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 232–245, June 1990.

[19] J. Coplien. Curiously recurring template patterns. *C++ Report*, 7(2):24–27, February 1995.

[20] P. Cousot and R. 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.

[21] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 84–97, Tucson, Arizona, 1978.

[22] K. Czarnecki and U. W. Eisenecker. *Generative Programming—Towards a New Paradigm of Software Engineering*. Addison Wesley Longman, 2000.

[23] J. de Guzman and D. Nuffer. The Spirit Parsing Library: Inline parsing in C++. *C/C++ Users Journal*, 21(9):22–30, September 2003.

[24] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical report, COMPAQ Systems Research Center, 1998.

[25] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *International Symposium on Static Analysis (SAS)*, pages 194–212, July 2001.

[26] Edison Design Group C++ front end. `http://www.edg.com/`, 2002.

[27] M. Emami, R. Ghiya, and L. 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.

[28] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[29] A. M. Erosa and L. J. Hendren. Taming control flow: A structured approach to eliminating GOTO statements. In *ICCL*, pages 229–240, May 1994.

[30] ESC/Java: Extended Static Checking for Java. `http://research.compaq.com/SRC/esc/`, 2000.

[31] D. Evans, J. Guttag, J. Horning, and Y. M. 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.

[32] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[33] FACT! — Multiparadigm programming with C++. `http://www.kfa-juelich.de/zam/FACT/start/index.html`, 2003.

[34] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–241, 2002.

[35] B. Fomitchev. STLport. `http://www.stlport.org`, 2001.

[36] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM Press, 2002.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[38] GNU compiler collection. `http://www.gnu.org/software/gcc/`, 2004.

[39] S. Ghemawat, K. H. Randall, and D. J. Scales. Field analysis: Getting useful and low-cost interprocedural informa tion. *ACM SIGPLAN Notices*, 35(5):334–344, 2000.

[40] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–15, January 1996.

[41] Gimpel Software. Flexelint for C/C++. http://www.gimpel.com/html/flex.htm, 2004.

[42] D. Gregor and S. Schupp. Making the usage of STL safe. In J. Gibbons and J. 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.

[43] D. Gregor, S. Schupp, and D. R. Musser. Design patterns for library optimizations. In J. K. Davis, editor, *Proceedings International Conference on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'01) Tampa, FL, 2001*, 2001.

[44] D. Gregor, S. Schupp, and D. R. Musser. Design patterns for library optimization. *Scientific Programming*, 11(4):309–320, 2003.

[45] M. R. Haghighat. *Symbolic Analysis for Parallelizing Compilers*. Kluwer Academic Publishers, 1995.

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

[47] W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[48] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[49] C. S. Horstmann. Safe STL. `http://www.horstmann.com/safestl.html`, 1995.

[50] ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++. *Technical Report on C++ Performance*, August 2003. Doc. N1487=03-0070.

[51] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodology*, 11(2):256–290, 2002.

[52] D. Jackson. Object models as heap invariants. pages 247–268, 2003.

[53] J. Järvi, G. Powell, and A. Lumsdaine. The Lambda Library: unnamed functions in C++. *Software—Practice and Experience*, 33:259–291, 2003.

[54] J. Järvi, J. Willcock, H. Hinnant, and A. Lumsdaine. Function overloading based on arbitrary properties of types. *C/C++ Users Journal*, 21(6):25–32, 2003.

[55] M. Jazayeri, R. Loos, and D. Musser, editors. *Generic Programming: International Seminar, Dagstuhl Castle, Germany, 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, 2000.

[56] S. C. Johnson. Lint, a C program checker. Technical Report 65, AT&T Bell Laboratories, 1978.

[57] D. Kapur, D. R. Musser, and A. A. Stepanov. Tecton: A language for manipulating generic objects. In J. Staunstrup, editor, *Proceedings of a Workshop on Program Specification*, volume 134 of *LNCS*, pages 402–414, Aarhus, Denmark, Aug. 1981. Springer.

[58] S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. In *Proceedings of the ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 231–245. ACM Press, 2002.

[59] G. Kiczales, E. Hisdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 327–355, 2001.

[60] G. Kiczales, J. Lamping, A. Mendhekar, C. s Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwi. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242, 1997.

[61] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[62] A. Koenig and B. E. Moo. *Accelerated C++*. Addison-Wesley, 2000.

[63] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–278, 1995.

[64] D. Kühl, J. Järvi, J. Siek, and M. Marcus. Detecting valid expressions at compile time. Posting on C++ Boost mailing list, April 2002.

[65] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 1–16, January 2002.

[66] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, pages 177–190, August 2001.

[67] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 21–34, July 1988.

[68] G. T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In H. Kilov and W. Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 121–142. Kluwer Academic Publishers, 1996.

[69] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: Notations and tools supporting detailed design in Java. In *OOPSLA '00 Companion*, pages 105–106, 2000.

[70] V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 114–119, 1982.

[71] Metrowerks CodeWarrior. `http://www.metrowerks.com/`, 2004.

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

[73] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 1–11, 2002.

[74] M. Müller. Abstraction benchmarks and performance of C++ applications. In *Proceedings of the International Conference on Supercomputing*, 2000.

[75] D. Musser, S. Schupp, and R. Loos. Requirement oriented programming. In M. Jazayeri, R. Loos, and D. Musser, editors, *Generic Programming: International Seminar, Dagstuhl Castle, Germany, 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*, pages 12–24. Springer-Verlag, Heidelberg, Germany, 2000.

[76] D. Musser and A. Stepanov. *The ADA Generic Library: Linear List Processing Packages*. Springer-Verlag, 1989.

[77] D. Musser and A. Stepanov. Algorithm-oriented generic libraries. *Software-practice and experience*, 27(7):623–642, July 1994.

[78] D. R. Musser. The Tecton Concept Description Language. `http://www.cs.rpi.edu/~musser/gp/tecton/tecton1.ps.gz`, July 1998.

[79] D. R. Musser. Tecton description of STL container and iterator concept s. `http://www.cs.rpi.edu/~musser/gp/tecton/container.ps.gz`, August 1998.

[80] D. R. Musser. An algorithm concept web. `http://www.cs.rpi.edu/~musser/gp/algorithm-concepts/`, 2003.

[81] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library Second Edition*. Addison-Wesley, 2001.

[82] D. R. Musser and A. A. Stepanov. Generic programming. In P. Gianni, editor, *Symbolic and Algebraic Computation: International Symposium ISSAC 1988*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25. Springer-Verlag, 1989.

[83] N. Myers. A new and useful template technique. In *C++ Gems*. Cambridge University Press, December 1996.

[84] J. R. C. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 67–78, June 1995.

[85] 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.

[86] J. V. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. Tholburn. POOMA: A framework for scientific simulations on parallel architectures. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 553–594. MIT Press, 1996.

[87] A. D. Robison. The abstraction penalty for small objects in C++. In *POOMA '96: The Parallel Object-Oriented Methods and Applications Conference*, 1996.

[88] 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.

[89] A. Rountev. *Dataflow Analysis of Software Fragments*. PhD thesis, Rutgers University, August 2002.

[90] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. In *Proceedings of the 25th International Conference on Software Engineering*, pages 210–220. IEEE Computer Society, 2003.

[91] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 16–31, January 1996.

[92] R. Sakellariou. Symbolic evaluation of sums for parallelising compilers. In *IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, pages 685–690. Wissenschaft & Technik Verlag, 1997.

[93] S. Schupp, D. Gregor, and D. Musser. Algebraic concepts represented in C++. Technical Report TR-00-8, Rensselaer Polytechnic Institute, 2000.

[94] S. Schupp, D. Gregor, D. R. Musser, and S.-M. Liu. Library transformations. In *First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), Florence, Italy*, pages 109–121. IEEE, November 2001.

[95] S. Schupp, D. Gregor, D. R. Musser, and S.-M. Liu. User-extensible simplification–Type-based optimizer generators. In R. Wilhelm, editor, *Proceedings of the 10th International Conference on Compiler Construction*, number 2027 in LNCS, pages 86–101. Springer-Verlag, Berlin Heidelberg, April 2001.

[96] S. Schupp, D. Gregor, D. R. Musser, and S.-M. Liu. Semantic and behavioral library transformations. *Information and Software Technology*, 44(13):797–810, 2002.

[97] S. Schupp, D. Gregor, B. Osman, D. R. Musser, J. Siek, L.-Q. Lee, and A. Lumsdaine. Concept-based component libraries and optimizing compilers. Technical report, RPI Computer Science Department Technical Report 02-02, 2002.

[98] Standard Template Library programmer's guide. `http://www.sgi.com/tech/stl/`, 2004.

[99] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, August 2001.

[100] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice Hall, Englewood Cliffs, New Jersey, 1981.

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

[102] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User's Guide and Reference Manu al*. C++ In-Depth Series. Addison-Wesley, December 2002.

[103] J. G. Siek and A. Lumsdaine. The Matrix Template Library: Generic components for high-performance scientific computing. *Computing in Science and Engineering*, 1(6):70–78, Nov/Dec 1999.

[104] Y. Smaragdakis and B. McNamara. FC++: Functional tools for object-oriented tasks. *Software—Practice and Experience*, 32(10):1015–1033, 2002.

[105] Splint: Annotation-assisted lightweight static checking. `http://www.splint.org/`, 2004.

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

[107] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 132–143, 1977.

[108] R. A. van Engelen. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the International Conference on Compiler Construction*, number 2027 in LNCS, pages 118–132, April 2001.

[109] R. A. van Engelen and K. A. Gallivan. Tight non-linear loop timing estimation. In *Proceedings of the International Workshop on Innovative Architectures for Future Generation High-Performance Processors and Systems (IWIA)*, pages 21–26, Maui, Hawaii, January 2002.

[110] D. Vandevoorde and N. M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2002.

[111] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4), 1995.

[112] T. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University, (forthcoming).

[113] T. L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

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

[115] C. Verbrugge, P. Co, and L. Hendren. Generalized constant propagation: A study in C. In T. Gyimothy, editor, *Proceedings of the International Conference on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 74–90, April 1996.

[116] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[117] C. Wang and D. R. Musser. Dynamic verification of C++ generic algorithms. *Software Engineering*, 23(5):314–323, 1997.

[118] J. Wilander and M. Kamkar. A comparison of publicly available tools for static intrusion prevention. In *Proceedings of the Nordic Workshop on Secure IT Systems*, pages 68–84, Karlstad, Sweden, November 2002.

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

[120] R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In I. Rival, editor, *Ordered Sets*, pages 445–470. NATO Advanced Study Institute, September 1981.

[121] M. Wolfe. Beyond induction variables. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 162–174, 1992.

[122] L. Zolman. An STL error message decryptor for Visual C++. *C/C++ User's Journal*, 19(7):24–30, July 2001.

APPENDIX A
Example generic iterator specification

This appendix provides the generic iterator specification employed by STLlint as a class template `_Safe_iterator`. Instantiations of `_Safe_iterator` are used as the iterator types of container specifications, and these safe iterators provide the same syntactic interface as their underlying iterators but with Semple assertions to verify correct usage.

```
template<typename _Iterator, typename _Sequence>
  class _Safe_iterator
  {
    typedef _Safe_iterator _Self;

    typedef std::iterator_traits<_Iterator> _Traits;

  public:
    typedef typename _Traits::iterator_category iterator_category;
    typedef typename _Traits::value_type        value_type;
    typedef typename _Traits::difference_type   difference_type;
    typedef typename _Traits::reference         reference;
    typedef typename _Traits::pointer           pointer;

    // Construction and assignment
    _Safe_iterator();
    _Safe_iterator(unsigned int __position, const _Sequence* __seq);
    _Safe_iterator(const _Safe_iterator& __other);
    template<typename _Mutable>
      _Safe_iterator(const _Safe_iterator<_Mutable, _Sequence>& __other);
    _Safe_iterator& operator=(const _Safe_iterator& __other);

    // Dereference operations
    reference operator*() const;
    pointer operator->() const;

    // Input iterator operations
    _Safe_iterator& operator++();
    _Safe_iterator operator++(int);
```

```
// Bidirectional iterator operations
_Safe_iterator& operator--();
_Safe_iterator operator--(int);


// Random access iterator operations
reference operator[](const difference_type& __n) const;
_Safe_iterator& operator+=(const difference_type& __n);
_Safe_iterator operator+(const difference_type& __n) const;
_Safe_iterator& operator-=(const difference_type& __n)
_Safe_iterator operator-(const difference_type& __n) const;

// Iterator state queries
bool _M_singular() const
{ return (!_M_sequence || _M_sequence->_M_version != _M_version); }

bool _M_dereferenceable() const
  {
    return (_M_sequence && _M_position >= 0
            && _M_position < _M_sequence->_M_size);
  }

bool _M_past_the_end() const
  { return _M_position >= _M_sequence->_M_size; }

bool _M_incrementable() const { return _M_dereferenceable(); }
bool _M_decrementable() const { return _M_position > 0; }

bool _M_can_advance(const difference_type& __n) const
{ return (_M_position + __n >= 0
          && _M_position + __n <= _M_sequence->_M_size); }

reference _M_get_data() const
{
  return (_M_sequence? static_cast<reference>(_M_sequence->_M_data)
          : _M_any_data);
}


// Iterator invalidation
void _M_invalidate();

// Data members
const _Sequence* _M_sequence;
unsigned int     _M_version;
unsigned int     _M_position;
```

```
    static value_type _M_any_data;
  };

// Iterator comparison operators
#define __STLLINT_ITERATOR_OP(Op,Kind) /* see below */
__STLLINT_ITERATOR_OP(==,input)
__STLLINT_ITERATOR_OP(!=,input)
__STLLINT_ITERATOR_OP(<,random_access)
__STLLINT_ITERATOR_OP(<=,random_access)
__STLLINT_ITERATOR_OP(>,random_access)
__STLLINT_ITERATOR_OP(>=,random_access)
#undef __STLLINT_ITERATOR_OP
```

A.1 Support macros

Several macros provide error message text and markup for the assertions performed by STLlint's safe iterators.

```
#define __STLLINT_ITERATOR_ERROR(Op,Num,Name,State) "attempt to " #Op   \
  " a <font color=primary><arg num=" #Num " name=" #Name">" #State      \
  " iterator</>"

#define __STLLINT_ITERATOR_COMPARE_ERROR "attempt to compare <font "   \
  "color=primary><arg num=1 name=lhs><arg num=2 name=rhs>iterators</>" \
  "from different containers"

#define __STLLINT_ITERATOR_DIFFERENCE_ERROR "attempt to compute the" \
  " difference between <font color=primary><arg num=1 name=lhs><arg" \
  " num=2 name=rhs>iterators</> from different containers"
```

A.2 Construction and assignment

The default constructor creates a singular iterator, unattached to any sequence.

```
template<typename _Iterator, typename _Sequence>
_Safe_iterator<_Iterator, _Sequence>::_Safe_iterator()
  : _M_sequence(0), _M_version(0),_M_position(0) { }
```

The primary constructor for safe iterators, used by sequences (containers) to attach an iterator to that sequence and specify the position of the iterator within that sequence. This constructor may only be invoked to construct nonsingular iterators.

```
template<typename _Iterator, typename _Sequence>
_Safe_iterator<_Iterator, _Sequence>::
_Safe_iterator(unsigned int __position, const _Sequence* __seq)
  : _M_sequence(__seq), _M_version(__seq->_M_version),
    _M_position(__position) { }
```

The copy constructor and assignment operator for safe iterators copy the sequence and position of the incoming iterator, `other`. As a nonsingular iterator may be copied, we verify that the incoming iterator is nonsingular. The `copy_object` event permits customization that allows properties attached to iterators (such as the fact that an iterator is being returned from a `lower_bound` operation) to be propagated to any copies.

```
template<typename _Iterator, typename _Sequence>
_Safe_iterator<_Iterator, _Sequence>::
_Safe_iterator(const _Safe_iterator& __other)
  : _M_sequence(__other._M_sequence), _M_version(__other._M_version),
    _M_position(__other._M_position)
{
  _STLlint::event<std::__events::copy_object,
                  void(_Self&, const _Self&)> ep(*this, __other);
  _STLlint::assertion(!__other._M_singular(),
                      __STLLINT_ITERATOR_ERROR(copy,2,other,singular));
}

template<typename _Iterator, typename _Sequence>
_Safe_iterator<_Iterator, _Sequence>&
_Safe_iterator<_Iterator, _Sequence>::
operator=(const _Safe_iterator& __other);
{
  _STLlint::event<std::__events::copy_object,
                  _Self&(_Self&, const _Self&)> ep(*this, __other);
   _STLlint::assertion(!__other._M_singular(),
                      __STLLINT_ITERATOR_ERROR(copy,2,other,singular));
  _M_sequence = __other._M_sequence;
  _M_version = __other._M_version;
  _M_position = __other._M_position;
  return ep(*this);
}
```

Constant iterators can be constructed from mutable iterators via the converting copy constructor. With the exception of the `_Mutable` parameter, this construc-

tor is equivalent to a copy constructor.

```
template<typename _Iterator, typename _Sequence>
template<typename _Mutable>
_Safe_iterator<_Iterator, _Sequence>::
_Safe_iterator(const _Safe_iterator<_Mutable, _Sequence>& __other);

  : _M_sequence(__other._M_sequence), _M_version(__other._M_version),
    _M_position(__other._M_position)
{
  typedef _Safe_iterator<_Mutable, _Sequence> _Other;
  _STLlint::event<std::__events::copy_object,
                  void(_Self&, const _Other&)> ep(*this, __other);

  _STLlint::assertion(!__other._M_singular(),
                      __STLLINT_ITERATOR_ERROR(copy,2,other,singular));
}
```

A.3 Dereference operations

The dereference operators require the iterator to be dereferenceable. We check for singular and past-the-end iterators separately, so that we can provide more accurate diagnostics to the user.

```
template<typename _Iterator, typename _Sequence>
typename _Safe_iterator<_Iterator, _Sequence>::reference
_Safe_iterator<_Iterator, _Sequence>::operator*() const
{
  _STLlint::event<std::__events::dereference, reference(const _Self&)>
    ep(*this);
  _STLlint::assertion(!_M_singular(),
                      __STLLINT_ITERATOR_ERROR(dereference,1,this,
                                               singular));
   _STLlint::assertion(!_M_sequence || _M_dereferenceable(),
                      __STLLINT_ITERATOR_ERROR(dereference,1,this,
                                               past-the-end));
  return ep(_M_get_data());
}

template<typename _Iterator, typename _Sequence>
typename _Safe_iterator<_Iterator, _Sequence>::pointer
_Safe_iterator<_Iterator, _Sequence>::operator->() const
{
```

```
  _STLlint::event<std::__events::dereference, pointer(const _Self&)>
    ep(*this);
  _STLlint::assertion(!_M_singular(),
                      __STLLINT_ITERATOR_ERROR(dereference,1,this,
                                               singular));
  _STLlint::assertion(!_M_sequence || _M_dereferenceable(),
                      __STLLINT_ITERATOR_ERROR(dereference,1,this,
                                               past-the-end));
  return ep(&_M_get_data());
}
```

A.4 Input iterator operations

The iterator pre- and post-increment operations verify that the iterator is incrementable, first by ensuring that it is nonsingular and then by checking if it is not past-the-end.

```
template<typename _Iterator, typename _Sequence>
_Safe_iterator<_Iterator, _Sequence>&
_Safe_iterator<_Iterator, _Sequence>::operator++()
{
  _STLlint::assertion(!_M_singular(),
                      __STLLINT_ITERATOR_ERROR(increment,1,this,
                                               singular));
  _STLlint::assertion(!_M_sequence || _M_incrementable(),
                      __STLLINT_ITERATOR_ERROR(increment,1,this,
                                               past-the-end));
  ++_M_position;
  return *this;
}

template<typename _Iterator, typename _Sequence>
_Safe_iterator<_Iterator, _Sequence>
_Safe_iterator<_Iterator, _Sequence>::operator++(int)
{
  _STLlint::assertion(!_M_singular(),
                      __STLLINT_ITERATOR_ERROR(increment,1,this,
                                               singular));
  _STLlint::assertion(!_M_sequence || _M_incrementable(),
                      __STLLINT_ITERATOR_ERROR(increment,1,this,
                                               past-the-end));
  _Safe_iterator __tmp(*this);
  ++_M_position;
  return __tmp;
```

```
}
```

A.5 Bidirectional iterator operations

The iterator pre- and post-decrement operations verify that the iterator is decrementable, first by ensuring that it is nonsingular and then by checking its position against zero. The static assertion ensures that a program error occurs if the decrement operator is applied to a non-bidirectional iterator.

```
template<typename _Iterator, typename _Sequence>
_Safe_iterator<_Iterator, _Sequence>&
_Safe_iterator<_Iterator, _Sequence>::operator--()
{
  _STLLINT_STATIC_ASSERT((is_base_of<std::bidirectional_iterator_tag,
                                     iterator_category>::value));
  _STLlint::assertion(!_M_singular(),
                      __STLLINT_ITERATOR_ERROR(increment,1,this,
                                               singular));
  _STLlint::assertion(!_M_sequence || _M_decrementable(),
                      __STLLINT_ITERATOR_ERROR(increment,1,this,
                                               start-of-sequence));
  --_M_position;
  return *this;
}

template<typename _Iterator, typename _Sequence>
_Safe_iterator<_Iterator, _Sequence>
_Safe_iterator<_Iterator, _Sequence>::operator--(int)
{
  _STLLINT_STATIC_ASSERT((is_base_of<std::bidirectional_iterator_tag,
                                     iterator_category>::value));
  _STLlint::assertion(!_M_singular(),
                      __STLLINT_ITERATOR_ERROR(increment,1,this,
                                               singular));
  _STLlint::assertion(!_M_sequence || _M_decrementable(),
                      __STLLINT_ITERATOR_ERROR(increment,1,this,
                                               start-of-sequence));
  _Safe_iterator __tmp(*this);
  --_M_position;
  return __tmp;
}
```

A.6 Random access iterator operations

The subscripting operator provides constant-time access to an element any number of steps forward (or backward) from the element this iterator references. It verifies that the iterator is nonsingular and checks that the subscripting is within bounds for this iterator.

```
template<typename _Iterator, typename _Sequence>
typename _Safe_iterator<_Iterator, _Sequence>::reference
_Safe_iterator<_Iterator, _Sequence>::
operator[](const difference_type& __n) const
{
  _STLLINT_STATIC_ASSERT((is_base_of<std::random_access_iterator_tag,
                                   iterator_category>::value));
  _STLlint::assertion(!_M_singular(),
                      __STLLINT_ITERATOR_ERROR(subscript,1,this,
                                               singular));
  _STLlint::assertion(!_M_sequence || _M_can_advance(__n),
    "<font color=primary><arg num=2 name=n>subscript</> out-of-range"
    " for <font color=secondary><arg num=1 name=this>iterator</>");
  return _M_sequence->_M_data;
}
```

Random access iterators can be moved any number of steps in constant time, but singular iterators cannot be moved and nonsingular iterators cannot be moved past the beginning of the sequence or beyond the past-the-end position.

```
template<typename _Iterator, typename _Sequence>
_Safe_iterator<_Iterator, _Sequence>&
_Safe_iterator<_Iterator, _Sequence>::
operator+=(const difference_type& __n)
{
  _STLLINT_STATIC_ASSERT((is_base_of<std::random_access_iterator_tag,
                                   iterator_category>::value));
  _STLlint::assertion(!_M_singular(),
                      __STLLINT_ITERATOR_ERROR(advance,1,this,
                                               singular));
  _STLlint::assertion(!_M_sequence || _M_can_advance(__n),
    "attempt to advance an <font color=secondary><arg num=1"
    " name=this> iterator</> outside its valid range");
  _M_position += __n;
  return *this;
```

```
}

template<typename _Iterator, typename _Sequence>
_Safe_iterator<_Iterator, _Sequence>
_Safe_iterator<_Iterator, _Sequence>::
operator+(const difference_type& __n) const
{
  _STLLINT_STATIC_ASSERT((is_base_of<std::random_access_iterator_tag,
                                    iterator_category>::value));
  _STLlint::assertion(!_M_singular(),
                      __STLLINT_ITERATOR_ERROR(advance,1,this,
                                               singular));
  _STLlint::assertion(!_M_sequence || _M_can_advance(__n),
    "attempt to advance an <font color=secondary><arg num=1"
    " name=this>iterator</> outside its valid range");
  _Safe_iterator __tmp(*this);
  __tmp._M_position += __n;
  return __tmp;
}

template<typename _Iterator, typename _Sequence>
_Safe_iterator<_Iterator, _Sequence>&
_Safe_iterator<_Iterator, _Sequence>::
operator-=(const difference_type& __n)
{
  _STLLINT_STATIC_ASSERT((is_base_of<std::random_access_iterator_tag,
                                    iterator_category>::value));
  _STLlint::assertion(!_M_singular(),
                      __STLLINT_ITERATOR_ERROR(retreat,1,this,
                                               singular));
  _STLlint::assertion(_M_can_advance(-__n),
    "attempt to retreat an <font color=secondary><arg num=1"
    " name=this>iterator</> outside its valid range");
  _M_position -= __n;
  return *this;
}

template<typename _Iterator, typename _Sequence>
_Safe_iterator<_Iterator, _Sequence>
_Safe_iterator<_Iterator, _Sequence>::
operator-(const difference_type& __n) const
{
  _STLLINT_STATIC_ASSERT((is_base_of<std::random_access_iterator_tag,
                                    iterator_category>::value));
  _STLlint::assertion(!_M_singular(),
                      __STLLINT_ITERATOR_ERROR(retreat,1,this,
```

```
                                             singular));
  _STLlint::assertion(_M_can_advance(-__n),
    "attempt to retreat an <font color=secondary><arg num=1"
    " name=this>iterator</> outside its valid range");
  _Safe_iterator __tmp(*this);
  __tmp._M_position -= __n;
  return __tmp;
}
```

A.7 Iterator invalidation

Iterator invalidation involves determining all iterators that may reference the same element as this iterator and invalidating them by setting the version value to zero.

```
template<typename _Iterator, typename _Sequence>
void _Safe_iterator<_Iterator, _Sequence>::_M_invalidate()
{
  STLLINT_FOR_EACH(_Self __i) {
    if (__i._M_sequence == _M_sequence && __i._M_position == _M_position)
      __i._M_version = 0;
  }
}
```

A.8 Iterator comparison operators

Iterator comparison operators ensure that neither operand is a singular iterator and that both operands refer to the same sequence, thus checking the domain of the operation. The result of the operations, `==`, `!=`, `<`, `<=`, `>`, and `>=`, is the result of performing the same comparison on the underlying iterator position.

```
#define __STLLINT_ITERATOR_OP(Op,Kind)                                   \
template<typename _IteratorL, typename _IteratorR, typename _Sequence>  \
inline bool                                                              \
operator Op(const _Safe_iterator<_IteratorL, _Sequence>& __lhs,         \
            const _Safe_iterator<_IteratorR, _Sequence>& __rhs)         \
{                                                                        \
   typedef typename std::iterator_traits<_IteratorL>::iterator_category \
     Category;                                                           \
   _STLLINT_STATIC_ASSERT((is_base_of<std::Kind##_iterator_tag,         \
```

```
                                    Category>::value));               \
  _STLlint::assertion(!__lhs._M_singular(),                           \
                      __STLLINT_ITERATOR_ERROR(compare,1,lhs,singular));\
  _STLlint::assertion(!__rhs._M_singular(),                           \
                      __STLLINT_ITERATOR_ERROR(compare,2,rhs,singular));\
  _STLlint::assertion(__lhs._M_sequence == __rhs._M_sequence,         \
                      __STLLINT_ITERATOR_COMPARE_ERROR);              \
  return __lhs._M_position Op __rhs._M_position;                      \
}                                                                     \
                                                                      \
template<typename _Iterator, typename _Sequence>                      \
inline bool                                                           \
operator Op(const _Safe_iterator<_Iterator, _Sequence>& __lhs,        \
             const _Safe_iterator<_Iterator, _Sequence>& __rhs)       \
{                                                                     \
   typedef typename std::iterator_traits<_Iterator>::iterator_category  \
      Category;                                                       \
   _STLLINT_STATIC_ASSERT((is_base_of<std::Kind##_iterator_tag,       \
                                    Category>::value));               \
  _STLlint::assertion(!__lhs._M_singular(),                           \
                      __STLLINT_ITERATOR_ERROR(compare,1,lhs,singular));\
  _STLlint::assertion(!__rhs._M_singular(),                           \
                      __STLLINT_ITERATOR_ERROR(compare,2,rhs,singular));\
  _STLlint::assertion(__lhs._M_sequence == __rhs._M_sequence,         \
                      __STLLINT_ITERATOR_COMPARE_ERROR);              \
  return __lhs._M_position Op __rhs._M_position;                      \
}
```

APPENDIX B

Example container specification: STL vector

The specification for the STL `vector` container, implemented as a class template, involves interactions with safe iterators and complex iterator invalidation semantics. The following is an abridged specification of the STL `vector` as it appears in STLlint.

```
template<typename _Tp, typename _Allocator = std::allocator<_Tp> >
  class vector
  : public _STLlint::_Safe_sequence<_Tp, vector<_Tp, _Allocator> >
  {
    typedef _STLlint::_Safe_sequence<_Tp, vector>     _Base;
    typedef iterator<random_access_iterator_tag, _Tp> _Base_iterator;

  public:
    typedef _Tp&                                      reference;
    typedef const _Tp&                                const_reference;
    typedef _STLlint::_Safe_iterator<_Base_iterator,vector>
                                                      iterator;
    typedef size_t                                    size_type;
    typedef ptrdiff_t                                 difference_type;
    typedef _Tp                                       value_type;
    typedef _Allocator                                allocator_type;
    typedef typename _Allocator::pointer              pointer;
    typedef typename _Allocator::const_pointer        const_pointer;

    // Construction and destruction
    explicit vector(const _Allocator& = _Allocator())
    : _M_guaranteed_capacity(0) { }

    explicit vector(size_type __n, const _Tp& __value = _Tp(),
                    const _Allocator& = _Allocator());

    template <class _InputIterator>
      vector(_InputIterator __first, _InputIterator __last,
             typename _STLlint::enable_if<
               (!_STLlint::__is_integral<_InputIterator>::value),
               const _Allocator&>::type = _Allocator());

    vector(const vector<_Tp,_Allocator>& __x);

    ~vector();
```

```
// Assignment
vector<_Tp,_Allocator>&
operator=(const vector<_Tp,_Allocator>& __x);

template <class _InputIterator>
  typename _STLlint::enable_if<
    (!_STLlint::__is_integral<_InputIterator>::value)>::type
  assign(_InputIterator __first, _InputIterator __last);

void assign(size_type __n, const _Tp& __x);

allocator_type get_allocator() const { return allocator_type(); }

// iterators:
iterator begin() { return iterator(0, this); }
iterator end() { return iterator(this->_M_size, this); }

// Capacity:
size_type size() const { return this->_M_size; }
size_type max_size() const { return size_type(-1) / sizeof(_Tp); }
void resize(size_type __sz, _Tp __c = _Tp());
size_t capacity() const { return _M_guaranteed_capacity; }
bool empty() const { return this->_M_size == 0; }
void reserve(size_type __n);

// element access:
reference operator[](size_type __n)
{
  __stllint_check_subscript(__n);
  return this->_M_data;
}

const_reference operator[](size_type __n) const
{
  __stllint_check_subscript(__n);
  return this->_M_data;
}

// Modifiers
void push_back(const _Tp& __x);
void pop_back();
iterator insert(iterator __position, const _Tp& __x);
void insert(iterator __position, size_type __n, const _Tp& __x);

template <class _InputIterator>
```

```
      typename _STLlint::enable_if<
        (!_STLlint::__is_integral<_InputIterator>::value)>::type
      insert(iterator __position,
             _InputIterator __first, _InputIterator __last);

    iterator erase(iterator __position);
    iterator erase(iterator __first, iterator __last);

    void swap(vector<_Tp,_Allocator>& __x)
    {
      _Base::_M_swap(*this, __x);
      std::swap(_M_guaranteed_capacity, __x._M_guaranteed_capacity);
    }

    void clear()
    {
      this->_M_size = 0;
      this->_M_invalidate_all();
      _M_guaranteed_capacity = 0;
    }

  private:
    size_type _M_guaranteed_capacity;

    void _M_update_guaranteed_capacity()
    {
      if (this->_M_size > _M_guaranteed_capacity) {
        _M_guaranteed_capacity = _STLlint::range(this->_M_size,
                                                 max_size());
      }
    }
  };
```

B.1 Safe sequence base class

The `_Safe_sequence` base class employs the Curiously Recurring Template pattern [19] to provide a functionality common to all STL container specifications. It is reproduced here in part to aid in the exposition of the `vector` class template.

```
template<typename _Tp, typename _Sequence>
  class _Safe_sequence
  {
  public:
    explicit _Safe_sequence(unsigned int __size = 0,
```

```
                            const _Tp& __data = _Tp())
      : _M_size(__size), _M_data(__data), _M_version(1) { }

    // Iterator invalidation
    void _M_invalidate_all();
    void _M_invalidate_from(size_type __n);

    // Data members
    _STLlint::unsigned_int _M_size;
    mutable _Tp _M_data;
    mutable _STLlint::unsigned_int _M_version;
  };
```

B.1.1 Iterator invalidation

`_Base_sequence` provides several methods that invalidate certain iterators by either updating the version number of the sequence or by setting the version number of iterators that meet certain requirements to zero.

```
template<typename _Tp, typename _Sequence>
void _Safe_sequence<_Tp, _Sequence>::_M_invalidate_all()
{
  _M_version++;
}

template<typename _Tp, typename _Sequence>
void _Safe_sequence<_Tp, _Sequence>::_M_invalidate_from(size_type __n)
{
  typedef typename _Sequence::iterator iterator;
  _Sequence* __self = static_cast<_Sequence*>(this);
  STLLINT_FOR_EACH(iterator __i) {
    if (__i._M_sequence == __self && __i._M_position >= __n)
      __i._M_version = 0;
  }
}
```

B.2 Construction and destruction

The `vector` constructors initialize the underlying safe sequence to the size of the incoming sequence, iterator range, or value and initialize the value accordingly. Of particular interest is the use of `enable_if` in the second constructor, which

disambiguates it from the first constructor in the case where the value type and size type are equivalent [5, §23.1.1/9] by eliminating the latter from the overload set [54].

```
template<typename _Tp, typename _Allocator>
vector<_Tp, _Allocator>::
vector(size_type __n, const _Tp& __value = _Tp(),
       const _Allocator& = _Allocator())
  : _Base(__n, __value), _M_guaranteed_capacity(__n) { }

template<typename _Tp, typename _Allocator>
template <class _InputIterator>
vector<_Tp, _Allocator>::
vector(_InputIterator __first, _InputIterator __last,
       typename _STLlint::enable_if<
         (!_STLlint::__is_integral<_InputIterator>::value),
         const _Allocator&>::type = _Allocator())
  : _Base(_STLlint::__distance(__first, __last)),
    _M_guaranteed_capacity(0)
{
  __stllint_check_valid_range(__first, __last);
  if (__first != __last)
    this->_M_data = *__first;
  _M_update_guaranteed_capacity();
}

template<typename _Tp, typename _Allocator>
vector<_Tp, _Allocator>::vector(const vector<_Tp,_Allocator>& __x)
  : _Base(__x.begin(), __x.end()), _M_guaranteed_capacity(0)
{ _M_update_guaranteed_capacity(); }
```

The destructor implicitly destroys the `_Safe_sequence` base subobject, which increments the container version number and therefore invalidates all iterators referencing this `vector`.

```
template<typename _Tp, typename _Allocator>
  vector<_Tp, _Allocator>::~vector() { }
```

B.3 Assignment

Assignment to a `vector`, either via the copy assignment operator or via the `assign` member function, invalidates all iterators and overwrites the size and data

associated with this container based on the incoming container or sequence. Again, `enable_if` is employed to disambiguate the integral iterator case.

```
template<typename _Tp, typename _Allocator>
vector<_Tp,_Allocator>&
vector<_Tp, _Allocator>::operator=(const vector<_Tp,_Allocator>& __x)
{
  *static_cast<_Base*>(this) = __x;
  this->_M_invalidate_all();
  _M_update_guaranteed_capacity();
  return *this;
}

template<typename _Tp, typename _Allocator>
void
vector<_Tp, _Allocator>::assign(size_type __n, const _Tp& __x)
{
  this->_M_size = __n;
  this->_M_data = __x;
  this->_M_invalidate_all();
  _M_update_guaranteed_capacity();
}

template<typename _Tp, typename _Allocator>
template <class _InputIterator>
typename _STLlint::enable_if<
  (!_STLlint::__is_integral<_InputIterator>::value)>::type
vector<_Tp, _Allocator>::
assign(_InputIterator __first, _InputIterator __last)
{
  __stllint_check_valid_range(__first, __last);
  this->_M_size = _STLlint::__distance(__first, __last);
  if (__first != __last) this->_M_data = *__first;
  this->_M_invalidate_all();
  _M_update_guaranteed_capacity();
}
```

B.4 Capacity

The resizing operation changes the number of elements in the `vector`. When this number of elements exceeds the current capacity of the `vector`, reallocation must occur and all iterators are invalidated. If the new size is smaller than the old size, every iterator beyond the new size is invalidated as the container is shrunk.

```
template<typename _Tp, typename _Allocator>
void vector<_Tp,_Allocator>::resize(size_type __sz, _Tp __c = _Tp())
{
  if (this->_M_size == 0 || _STLlint::flip())
    this->_M_data = __c;

  if (__sz > _M_guaranteed_capacity) {
    this->_M_invalidate_all();
    this->_M_size = __sz;
    _M_update_guaranteed_capacity();
  }
  else if (__sz < this->_M_size) {
    this->_M_invalidate_from(__sz);
  }

  this->_M_size = __sz;
}
```

The `reserve` member function guarantees a specific capacity, invalidating all iterators if reallocation is required to achieve that capacity.

```
template<typename _Tp, typename _Allocator>
void vector<_Tp,_Allocator>::reserve(size_type __n);
{
  if (__n > _M_guaranteed_capacity) {
    _M_guaranteed_capacity = __n;
    this->_M_invalidate_all();
  }
}
```

B.5 Modifiers

The `push_back` operation adds an element at the end of the `vector`, which may require invalidation of all iterators due to reallocation.

```
template<typename _Tp, typename _Allocator>
void vector<_Tp,_Allocator>::push_back(const _Tp& __x);
{
  _STLlint::event<__events::push_back, void(vector&, const _Tp&)>
    ep(*this, __x);

  if (this->_M_size == 0 || _STLlint::flip())
```

```
    this->_M_data = __x;

  if (this->_M_size >= _M_guaranteed_capacity)
    this->_M_invalidate_all();

  ++this->_M_size;
  _M_update_guaranteed_capacity();
}
```

The `pop_back` operation verifies that the sequence is not empty, and removes the last element from the sequence. This operation invalidates the prior past-the-end iterator, only.

```
template<typename _Tp, typename _Allocator>
void vector<_Tp,_Allocator>::pop_back();
{
  __stllint_check_nonempty();

  _STLlint::event<__events::pop_back, void(vector&)> ep(*this);

  iterator __victim = end() - 1;
  __victim._M_invalidate();
  --this->_M_size;
}
```

Insertion of one or more elements into a `vector` requires that the given iterator `__position` be a nonsingular iterator that references this particular `vector`. All elements following the position of insertion must be shifted, invalidating any iterators to that point. Furthermore, if the guaranteed capacity is exceeded by inserting these elements, all iterators into this `vector` will be invalidated. The range insertion version also requires verification that the incoming iterator range is a valid range.

```
template<typename _Tp, typename _Allocator>
typename vector<_Tp,_Allocator>::iterator
vector<_Tp,_Allocator>::insert(iterator __position, const _Tp& __x)
{
  __stllint_check_insert(__position);

  _STLlint::event<__events::insert,
                  iterator(vector&, iterator, const _Tp&)>
    ep(*this, __position, __x);
```

```
  if (this->_M_size + 1 > _M_guaranteed_capacity)
    this->_M_invalidate_all();
  else
    this->_M_invalidate_from(__position._M_position);

  if (this->_M_size == 0 || _STLlint::flip())
    this->_M_data = __x;

  this->_M_size += 1;
  _M_update_guaranteed_capacity();
  return ep(iterator(__position._M_position, this));
}

template<typename _Tp, typename _Allocator>
void
vector<_Tp,_Allocator>::
insert(iterator __position, size_type __n, const _Tp& __x)
{
  __stllint_check_insert(__position);

  if (this->_M_size + __n > _M_guaranteed_capacity)
    this->_M_invalidate_all();
  else
    this->_M_invalidate_from(__position._M_position);

  if (this->_M_size == 0 || _STLlint::flip())
    this->_M_data = __x;

  this->_M_size += __n;
  _M_update_guaranteed_capacity();
}

template<typename _Tp, typename _Allocator>
template <class _InputIterator>
typename _STLlint::enable_if<
  (!_STLlint::__is_integral<_InputIterator>::value)>::type
vector<_Tp,_Allocator>::
insert(iterator __position,
       _InputIterator __first, _InputIterator __last)
{
  __stllint_check_insert_range(__position, __first, __last);

  size_t __n = _STLlint::__distance(__first, __last);
  if (this->_M_size + __n > _M_guaranteed_capacity)
    this->_M_invalidate_all();
```

```
  else
    this->_M_invalidate_from(__position._M_position);

  if (__first != __last && (this->_M_size == 0 || _STLlint::flip()))
    this->_M_data = *__first;

  this->_M_size += __n;
  _M_update_guaranteed_capacity();
}
```

Erasure from a `vector` requires that the iterators denoting the element(s) to be erased be nonsingular iterators that reference this particular `vector` instance. The `erase` member functions invalidate all elements from the point of erasure on, because elements must be shifted in memory.

```
template<typename _Tp, typename _Allocator>
typename vector<_Tp,_Allocator>::iterator
vector<_Tp,_Allocator>::erase(iterator __position)
{
  __stllint_check_erase(__position);
  this->_M_invalidate_from(__position._M_position);
  --this->_M_size;
  return iterator(__position._M_position, this);
}

template<typename _Tp, typename _Allocator>
typename vector<_Tp,_Allocator>::iterator
vector<_Tp,_Allocator>::erase(iterator __first, iterator __last)
{
  // __STLLINT_RESOLVE_LIB_DEFECTS
  // 151. can't currently clear() empty container
  __stllint_check_erase_range(__first, __last);

  _STLlint::event<__events::erase,
                  iterator(vector&, iterator, iterator)>
    ep(*this, __first, __last);

  this->_M_invalidate_from(__first._M_position);
  this->_M_size -= _STLlint::__distance(__first, __last);
  return ep(iterator(__first._M_position, this));
}
```

APPENDIX C
Experimental Results

Table C.1 illustrates the execution time of STLlint's static analysis along with the time spent by the Edison C++ front end [26] parsing and type-checking the program. The table also provides the ratio of analysis time to front end time, which ideally should be less than one.

Table C.1: STLlint's execution time on the STLlint test suite

Test	**Analysis time (seconds)**	**Front end time (seconds)**	**Ratio**
AccC++/chap0/hello.cc	0	0.43	0.000
AccC++/chap1/frame.cc	0.03	0.43	0.070
AccC++/chap1/greet.cc	0.01	0.43	0.023
AccC++/chap2/frame.cc	0.09	0.43	0.209
AccC++/chap3/avg.cc	0.02	0.45	0.044
AccC++/chap3/med.cc	0.35	0.79	0.443
AccC++/chap4/main1.cc	0.42	0.8	0.525
AccC++/chap4/main2.cc	1.57	1.17	1.342
AccC++/chap5/fails_iters.cc	3.46	1.2	2.883
AccC++/chap5/fails_iters_bad.cc	3.48	1.18	2.949
AccC++/chap5/fails_list.cc	3.86	0.88	4.386
AccC++/chap5/fails_list2.cc	3.84	0.88	4.364
AccC++/chap5/fails_vec1.cc	3.06	1.11	2.757
AccC++/chap5/fails_vec2.cc	3.13	1.16	2.698
AccC++/chap5/fails_vec2_bad.cc	3.37	1.19	2.832
AccC++/chap6/ext_fails1.cc	2.36	1.44	1.639
AccC++/chap6/ext_fails2.cc	2.09	1.62	1.290
AccC++/chap6/grade_analysis.cc	1.54	1.1	1.400
AccC++/chap6/palin.cc	0.52	0.55	0.945
AccC++/chap6/split_main.cc	5.08	0.59	8.610

Table C.1: STLlint's execution time... (*continued*)

Test	**Analysis time (seconds)**	**Front end time (seconds)**	**Ratio**
AccC++/chap7/grammar.cc	2.75	0.57	4.825
AccC++/chap7/wc.cc	0.4	0.48	0.833
AccC++/chap7/xref.cc	2.32	0.58	4.000
archetypes/adjacent_find.cpp	0.58	0.52	1.115
archetypes/find.cpp	0.38	0.55	0.691
archetypes/max_element.cpp	0.95	0.47	2.021
austern.cpp	0.54	0.61	0.885
austern2.cpp	0.53	0.62	0.855
austern3.cpp	0.61	0.62	0.984
back_insert_ok.cpp	0.5	0.58	0.862
classify/erase_remove.cpp	0.77	0.7	1.100
classify/erase_remove_ok.cpp	0.87	0.74	1.176
classify/heap.cpp	0.69	0.67	1.030
classify/heap_pop_pop.cpp	0.52	0.62	0.839
classify/heap_sort.cpp	0.78	0.62	1.258
classify/heap_with_sort_ok.cpp	1.3	0.83	1.566
classify/remove_fill.cpp	0.56	0.67	0.836
classify/remove_noaccess.cpp	0.8	0.73	1.096
classify/remove_overwrite.cpp	0.59	0.68	0.868
classify/sort.cpp	0.63	0.89	0.708
classify/sort_insert.cpp	0.78	0.87	0.897
classify/sort_insert2_ok.cpp	0.99	0.93	1.065
classify/sort_insert_intdiv.cpp	0.72	0.91	0.791
classify/sort_insert_ok.cpp	0.87	0.88	0.989
classify/sort_ok.cpp	0.57	0.87	0.655
classify/sort_partial_ok.cpp	0.7	0.8	0.875
classify/sort_slow_search.cpp	0.72	0.81	0.889
deque/cons/2.cpp	0.18	0.51	0.353

Table C.1: STLlint's execution time... (*continued*)

Test	**Analysis time (seconds)**	**Front end time (seconds)**	**Ratio**
deque/invalidation/1.cpp	0.1	0.47	0.213
deque/invalidation/2.cpp	0.1	0.47	0.213
deque/invalidation/3.cpp	0.19	0.46	0.413
deque/invalidation/4.cpp	0.21	0.46	0.457
deque/operators/1.cpp	0.04	0.44	0.091
erase_iterator_loop.cpp	0.6	0.47	1.277
find.cpp	0.36	0.58	0.621
find2-hard.cpp	0.46	0.55	0.836
find2.cpp	0.54	0.57	0.947
foreach.cpp	0.01	0.09	0.111
guarded_advance.cpp	0.33	0.52	0.635
list/austern-list.cpp	0.47	0.57	0.825
list/list_capacity.cpp	0.09	0.45	0.200
list/list_clear.cpp	0.05	0.43	0.116
list/list_compare_singular.cpp	0.01	0.43	0.023
list/list_ctor.cpp	0.55	0.5	1.100
list/list_deref_singular.cpp	0.04	0.46	0.087
list/list_erase_iter_loop.cpp	0.7	0.49	1.429
list/list_inc_singular.cpp	0.01	0.45	0.022
list/list_inval.cpp	1.28	0.5	2.560
list/list_loop.cpp	0.05	0.45	0.111
list/list_loop_singular.cpp	0.05	0.45	0.111
list/list_modifiers.cpp	3.62	0.5	7.240
list/list_postinc_singular.cpp	0.05	0.46	0.109
map/austern-map.cpp	0.87	0.61	1.426
map/init_map.cpp	0.21	0.48	0.438
map/insert/1.cpp	0.16	0.47	0.340
map/invalidation/1.cpp	0.11	0.49	0.224

Table C.1: STLlint's execution time… (*continued*)

Test	**Analysis time (seconds)**	**Front end time (seconds)**	**Ratio**
map/invalidation/2.cpp	0.92	0.52	1.769
map/invalidation/2_orig.cpp	1.05	0.51	2.059
map/invalidation/2_unrolled.cpp	2.7	0.5	5.400
map/map_deref_singular.cpp	0.04	0.46	0.087
map/operators/1.cpp	0.39	0.48	0.812
maybe_insert.cpp	1.13	0.9	1.256
memory/new_2delete.cpp	0.01	0.09	0.111
memory/new_delete.cpp	0	0.09	0.000
memory/new_delete_destruct.cpp	0.01	0.09	0.111
memory/new_leak.cpp	0	0.09	0.000
meyers.cpp	0.38	0.62	0.613
meyers_scary.cpp	0.47	0.63	0.746
multimap/invalidation/1.cpp	0.05	0.44	0.114
multimap/invalidation/2.cpp	0.56	0.49	1.143
multiset/insert/1.cpp	1.09	0.6	1.817
multiset/invalidation/2.cpp	0.55	0.48	1.146
return_ordering.cpp	0.01	0.09	0.111
set/invalidation/1.cpp	0.09	0.46	0.196
set/invalidation/2.cpp	0.64	0.51	1.255
set/invalidation/3.cpp	4.78	0.51	9.373
set/set_deref_singular.cpp	0.05	0.45	0.111
sort_different.cpp	0.23	0.74	0.311
sorted_vec_insert.cpp	0.94	0.88	1.068
transform_binary.cpp	0.47	0.59	0.797
unguarded_advance.cpp	0.36	0.49	0.735
vec_data_check.cpp	0.13	0.54	0.241
vector/bool/1.cpp	0.01	0.43	0.023
vector/capacity/1.cpp	0.02	0.44	0.045

Table C.1: STLlint's execution time... (*continued*)

Test	**Analysis time (seconds)**	**Front end time (seconds)**	**Ratio**
vector/cons/1.cpp	0.02	0.44	0.045
vector/cons/2.cpp	0.02	0.6	0.033
vector/cons/4.cpp	0.1	0.47	0.213
vector/element_access/1.cpp	0.02	0.43	0.047
vector/invalidation/1.cpp	0.1	0.47	0.213
vector/invalidation/2.cpp	0.11	0.48	0.229
vector/invalidation/3.cpp	0.35	0.48	0.729
vector/invalidation/4.cpp	0.18	0.48	0.375
vector/modifiers/1.cpp	0.04	0.44	0.091

Table C.2 provides static measures of the size of each test case, once the program has been transformed into Semple and the implementations of library components have been replaced with the corresponding (Semple) specifications. The "Fields" and "APL" columns refer to the maximum number of fields and the maximum access path length, respectively, within each program.

Table C.2: Static measures of Semple program size and complexity in the STLlint test suite

Test	**Statements**	**Functions**	**Fields**	**APL**
AccC++/chap0/hello.cc	19	3	24	0
AccC++/chap1/frame.cc	376	20	24	0
AccC++/chap1/greet.cc	115	11	24	0
AccC++/chap2/frame.cc	301	19	24	0
AccC++/chap3/avg.cc	239	18	24	0
AccC++/chap3/med.cc	4580	394	24	5
AccC++/chap4/main1.cc	4748	410	24	5
AccC++/chap4/main2.cc	9213	794	24	5
AccC++/chap5/fails_iters.cc	10676	914	24	5

Table C.2: Static measures of program size... (*continued*)

Test	**Statements**	**Functions**	**Fields**	**APL**
AccC++/chap5/fails_iters_bad.cc	10668	914	24	5
AccC++/chap5/fails_list.cc	7760	664	24	5
AccC++/chap5/fails_list2.cc	7752	664	24	5
AccC++/chap5/fails_vec1.cc	9346	799	24	5
AccC++/chap5/fails_vec2.cc	9369	800	24	5
AccC++/chap5/fails_vec2_bad.cc	9378	801	24	5
AccC++/chap6/ext_fails1.cc	13748	1137	24	5
AccC++/chap6/ext_fails2.cc	13323	1126	24	5
AccC++/chap6/grade_analysis.cc	6130	509	24	5
AccC++/chap6/palin.cc	2188	190	24	5
AccC++/chap6/split_main.cc	5515	446	24	5
AccC++/chap7/grammar.cc	5015	417	24	5
AccC++/chap7/wc.cc	1817	153	24	5
AccC++/chap7/xref.cc	4129	346	24	5
archetypes/adjacent_find.cpp	4786	388	3	5
archetypes/find.cpp	4558	367	4	5
archetypes/max_element.cpp	2153	181	3	5
austern.cpp	6012	484	4	5
austern2.cpp	6066	486	4	5
austern3.cpp	6112	490	24	5
back_insert_ok.cpp	5698	476	4	5
classify/erase_remove.cpp	8643	698	4	5
classify/erase_remove_ok.cpp	9581	762	4	5
classify/heap.cpp	8739	726	3	5
classify/heap_pop_pop.cpp	6020	503	3	5
classify/heap_sort.cpp	6054	503	3	5
classify/heap_with_sort_ok.cpp	14713	1183	5	5
classify/remove_fill.cpp	6055	499	3	5
classify/remove_noaccess.cpp	8643	698	4	5
classify/remove_overwrite.cpp	6491	514	3	5

Table C.2: Static measures of program size… (*continued*)

Test	Statements	Functions	Fields	APL
classify/sort.cpp	7963	639	5	5
classify/sort_insert.cpp	8938	704	5	5
classify/sort_insert2_ok.cpp	9596	760	5	5
classify/sort_insert_intdiv.cpp	8935	704	5	5
classify/sort_insert_ok.cpp	9571	760	5	5
classify/sort_ok.cpp	7957	638	5	5
classify/sort_partial_ok.cpp	8922	704	5	5
classify/sort_slow_search.cpp	8172	674	4	5
deque/cons/2.cpp	1997	147	3	5
deque/invalidation/1.cpp	1552	133	3	5
deque/invalidation/2.cpp	1444	125	3	5
deque/invalidation/3.cpp	1579	126	3	5
deque/invalidation/4.cpp	1651	129	3	5
deque/operators/1.cpp	283	10	3	5
erase_iterator_loop.cpp	2639	237	3	5
find.cpp	5498	445	24	5
find2-hard.cpp	5428	445	24	5
find2.cpp	5429	445	24	5
foreach.cpp	43	2	2	0
guarded_advance.cpp	925	74	3	5
list/austern-list.cpp	5495	430	4	5
list/list_capacity.cpp	933	74	3	5
list/list_clear.cpp	199	14	3	5
list/list_compare_singular.cpp	29	3	3	5
list/list_ctor.cpp	3100	225	3	5
list/list_deref_singular.cpp	680	61	3	5
list/list_erase_iter_loop.cpp	2140	185	3	5
list/list_inc_singular.cpp	31	3	3	5
list/list_inval.cpp	2935	209	3	5
list/list_loop.cpp	809	68	3	5

Table C.2: Static measures of program size... (*continued*)

Test	**Statements**	**Functions**	**Fields**	**APL**
list/list_loop_singular.cpp	770	66	3	5
list/list_modifiers.cpp	4038	272	3	5
list/list_postinc_singular.cpp	616	59	3	5
map/austern-map.cpp	5391	433	4	5
map/init_map.cpp	866	75	3	5
map/insert/1.cpp	1710	137	3	5
map/invalidation/1.cpp	973	81	3	5
map/invalidation/2.cpp	1852	149	3	5
map/invalidation/2_orig.cpp	1851	149	3	5
map/invalidation/2_unrolled.cpp	2035	149	3	5
map/map_deref_singular.cpp	680	61	3	5
map/operators/1.cpp	1678	138	24	5
maybe_insert.cpp	10247	818	5	5
memory/new_2delete.cpp	12	1	0	0
memory/new_delete.cpp	11	1	0	0
memory/new_delete_destruct.cpp	23	3	0	0
memory/new_leak.cpp	10	1	0	0
meyers.cpp	5878	477	4	5
meyers_scary.cpp	5943	481	24	5
multimap/invalidation/1.cpp	355	24	3	5
multimap/invalidation/2.cpp	1800	148	3	5
multiset/insert/1.cpp	5971	485	24	5
multiset/invalidation/2.cpp	1691	138	3	5
return_ordering.cpp	25	3	1	0
set/invalidation/1.cpp	906	73	3	5
set/invalidation/2.cpp	1791	141	3	5
set/invalidation/3.cpp	1992	141	3	5
set/set_deref_singular.cpp	680	61	3	5
sort_different.cpp	3770	320	3	5
sorted_vec_insert.cpp	8924	704	5	5

Table C.2: Static measures of program size... (*continued*)

Test	Statements	Functions	Fields	APL
transform_binary.cpp	5748	478	5	5
unguarded_advance.cpp	921	74	3	5
vec_data_check.cpp	646	61	3	3
vector/bool/1.cpp	36	4	3	5
vector/capacity/1.cpp	194	15	3	3
vector/cons/1.cpp	142	13	3	3
vector/cons/2.cpp	194	17	3	5
vector/cons/4.cpp	1410	90	3	5
vector/element_access/1.cpp	91	8	3	0
vector/invalidation/1.cpp	1601	137	3	5
vector/invalidation/2.cpp	1495	129	3	5
vector/invalidation/3.cpp	2801	203	4	5
vector/invalidation/4.cpp	2401	193	4	5
vector/modifiers/1.cpp	345	28	3	3

Table C.3 illustrates the analysis time required for each test case relative to the number of integer comparisons and variable replacements performed by STLlint. There is a correlation between an increase in the number of comparisons or replacements and increasing analysis time, which is better illustrated by Fig. 8.10.

Table C.3: Number of integer operations performed by STLlint relative to the analysis time

Test	Analysis time (seconds)	Comparisons	Replacements
AccC++/chap0/hello.cc	0	0	0
AccC++/chap1/frame.cc	0.03	57	10
AccC++/chap1/greet.cc	0.01	3	1
AccC++/chap2/frame.cc	0.09	1174	191
AccC++/chap3/avg.cc	0.02	9	2

Table C.3: Number of integer operations… (*continued*)

Test	**Analysis time (seconds)**	**Comparisons**	**Replacements**
AccC++/chap3/med.cc	0.35	635	79
AccC++/chap4/main1.cc	0.42	769	113
AccC++/chap4/main2.cc	1.57	12435	1165
AccC++/chap5/fails_iters.cc	3.46	30900	3848
AccC++/chap5/fails_iters_bad.cc	3.48	30175	3363
AccC++/chap5/fails_list.cc	3.86	38237	4772
AccC++/chap5/fails_list2.cc	3.84	41950	5138
AccC++/chap5/fails_vec1.cc	3.06	29227	2933
AccC++/chap5/fails_vec2.cc	3.13	29774	3446
AccC++/chap5/fails_vec2_bad.cc	3.37	31630	3920
AccC++/chap6/ext_fails1.cc	2.36	18103	1978
AccC++/chap6/ext_fails2.cc	2.09	14021	1527
AccC++/chap6/grade_analysis.cc	1.54	11306	1256
AccC++/chap6/palin.cc	0.52	1649	315
AccC++/chap6/split_main.cc	5.08	32013	6879
AccC++/chap7/grammar.cc	2.75	28654	3898
AccC++/chap7/wc.cc	0.4	3099	473
AccC++/chap7/xref.cc	2.32	22135	2728
archetypes/adjacent_find.cpp	0.58	503	95
archetypes/find.cpp	0.38	951	141
archetypes/max_element.cpp	0.95	341	96
austern.cpp	0.54	1227	220
austern2.cpp	0.53	1402	259
austern3.cpp	0.61	1488	281
back_insert_ok.cpp	0.5	1801	273
classify/erase_remove.cpp	0.77	2285	383
classify/erase_remove_ok.cpp	0.87	2611	435
classify/heap.cpp	0.69	1110	157

Table C.3: Number of integer operations… (*continued*)

Test	**Analysis time (seconds)**	**Comparisons**	**Replacements**
classify/heap_pop_pop.cpp	0.52	917	128
classify/heap_sort.cpp	0.78	1083	182
classify/heap_with_sort_ok.cpp	1.3	1469	235
classify/remove_fill.cpp	0.56	1426	276
classify/remove_noaccess.cpp	0.8	2402	414
classify/remove_overwrite.cpp	0.59	1475	280
classify/sort.cpp	0.63	1209	208
classify/sort_insert.cpp	0.78	1395	234
classify/sort_insert2_ok.cpp	0.99	3911	782
classify/sort_insert_intdiv.cpp	0.72	1587	281
classify/sort_insert_ok.cpp	0.87	2323	449
classify/sort_ok.cpp	0.57	1242	208
classify/sort_partial_ok.cpp	0.7	1407	224
classify/sort_slow_search.cpp	0.72	1385	256
deque/cons/2.cpp	0.18	220	28
deque/invalidation/1.cpp	0.1	16	0
deque/invalidation/2.cpp	0.1	96	8
deque/invalidation/3.cpp	0.19	563	61
deque/invalidation/4.cpp	0.21	744	91
deque/operators/1.cpp	0.04	48	0
erase_iterator_loop.cpp	0.6	5217	1030
find.cpp	0.36	889	162
find2-hard.cpp	0.46	2099	323
find2.cpp	0.54	1920	280
foreach.cpp	0.01	46	4
guarded_advance.cpp	0.33	3861	611
list/austern-list.cpp	0.47	696	169
list/list_capacity.cpp	0.09	434	59

Table C.3: Number of integer operations... (*continued*)

Test	**Analysis time (seconds)**	**Comparisons**	**Replacements**
list/list_clear.cpp	0.05	384	58
list/list_compare_singular.cpp	0.01	1	0
list/list_ctor.cpp	0.55	2371	570
list/list_deref_singular.cpp	0.04	1	0
list/list_erase_iter_loop.cpp	0.7	7865	1467
list/list_inc_singular.cpp	0.01	0	0
list/list_inval.cpp	1.28	11730	2286
list/list_loop.cpp	0.05	3	0
list/list_loop_singular.cpp	0.05	1	0
list/list_modifiers.cpp	3.62	23914	3178
list/list_postinc_singular.cpp	0.05	0	0
map/austern-map.cpp	0.87	6058	1048
map/init_map.cpp	0.21	2102	329
map/insert/1.cpp	0.16	576	86
map/invalidation/1.cpp	0.11	380	44
map/invalidation/2.cpp	0.92	10364	1828
map/invalidation/2_orig.cpp	1.05	11492	2080
map/invalidation/2_unrolled.cpp	2.7	35183	4643
map/map_deref_singular.cpp	0.04	1	0
map/operators/1.cpp	0.39	2995	532
maybe_insert.cpp	1.13	6236	1501
memory/new_2delete.cpp	0.01	2	0
memory/new_delete.cpp	0	2	0
memory/new_delete_destruct.cpp	0.01	2	0
memory/new_leak.cpp	0	2	0
meyers.cpp	0.38	300	80
meyers_scary.cpp	0.47	1078	205
multimap/invalidation/1.cpp	0.05	233	27

Table C.3: Number of integer operations... (*continued*)

Test	**Analysis time (seconds)**	**Comparisons**	**Replacements**
multimap/invalidation/2.cpp	0.56	6872	998
multiset/insert/1.cpp	1.09	9331	1768
multiset/invalidation/2.cpp	0.55	6773	989
return_ordering.cpp	0.01	1	0
set/invalidation/1.cpp	0.09	245	27
set/invalidation/2.cpp	0.64	6767	1016
set/invalidation/3.cpp	4.78	54540	9395
set/set_deref_singular.cpp	0.05	1	0
sort_different.cpp	0.23	9	0
sorted_vec_insert.cpp	0.94	3260	687
transform_binary.cpp	0.47	1081	205
unguarded_advance.cpp	0.36	4152	667
vec_data_check.cpp	0.13	1033	132
vector/bool/1.cpp	0.01	0	0
vector/capacity/1.cpp	0.02	21	2
vector/cons/1.cpp	0.02	6	1
vector/cons/2.cpp	0.02	4	0
vector/cons/4.cpp	0.1	68	4
vector/element_access/1.cpp	0.02	2	0
vector/invalidation/1.cpp	0.1	23	1
vector/invalidation/2.cpp	0.11	111	11
vector/invalidation/3.cpp	0.35	655	81
vector/invalidation/4.cpp	0.18	172	12
vector/modifiers/1.cpp	0.04	83	18