Report

Verifying C++ with STL Containers via Predicate Abstraction

Author(s):
Blanc, Nicolas; Kröning, Daniel; Groce, Alex

Publication Date:
2006

Permanent Link:
https://doi.org/10.3929/ethz-a-006788257

Rights / License:
In Copyright - Non-Commercial Use Permitted
Verifying C++ with STL Containers via Predicate Abstraction

Nicolas Blanc\textsuperscript{1}, Daniel Kroening\textsuperscript{1}, and Alex Groce\textsuperscript{2}

\textsuperscript{1} Computer Systems Institute, ETH Zürich
\textsuperscript{2} Laboratory for Reliable Software, Jet Propulsion Laboratory

Abstract. Verifying general properties of full-featured C++ code is beyond the scope of current model checking and predicate abstraction techniques. However, just as Microsoft’s SLAM project concentrates on verifying the usage of well-defined APIs in device drivers written in C, the restrictions the C++ Standard makes on the usage of the C++ Standard Template Library (STL) can be verified using a specialized form of counterexample-guided abstraction refinement. This paper describes a flexible and easily extensible predicate abstraction-based approach to the verification of STL usage. We formalize the semantics of the STL by means of a Hoare-style axiomatization. The verification requires an operational model, for which we show that it conservatively approximates the semantics given by the standard.

1 Introduction

C++ is one of the most widely used programming languages. Software programs including office applications, verification tools [1, 2], databases, games, and critical embedded systems are often implemented in C++. The language provides many useful features not provided by C, including support for object-oriented programming and generic programming, where general purpose algorithms or data structures can be applied to many types of data, with proper type-checking. Software model checking for C programs is widely recognized as providing real benefits for suitable programs, and is implemented by a number of tools [3, 4, 1, 2, 5]. All previous efforts to model check C++ code are based on explicit-state exploration and execution of the program; we propose to extend the popular predicate abstraction framework [6, 3] to the verification of C++ programs using abstract data types.

We concentrate our efforts on uses of the Standard Template Library (STL) [7], which provides a clear example of an advantage over verifying C code. Use of interesting data structures in C typically involves direct pointer manipulation and “hand-crafted” \textsuperscript{*} approaches to even common structures such as lists. Considerable effort must be spent in directly abstracting pointer behavior, not a strong suit of typical predicate abstraction engines. In contrast, code using the STL makes the operations explicit at the level of the data structure — STL

\textsuperscript{*} This research is supported by an award from IBM Research.
has made the most difficult part of the abstraction trivial, e.g., by replacing a
for loop stepping through next pointers of a struct with a for loop incrementing
an STL iterator into a list variable. Liskov and Zilles noted that abstract data
types (such as those provided by STL) allow programmers to abstract away from
the implementation details of commonly used structures and concentrate on the
task at hand [8]. We observe that abstract data types provide the same facility
in abstraction for verification tools.

We verify the use of STL calls rather than behavior for any particular
STL implementation. STL implementations are precisely the kind of pointer-
manipulation intensive, optimized-for-efficiency code that is difficult to abstract.
Choosing a particular implementation to verify would also be difficult. Additionally,
the STL implementations are typically well-tested, and even subtle bugs are
likely to be revealed given the large amount of code depending on correct behavior.
Discovering errors in a pattern of STL calls is therefore more useful to C++
programmers than verification of STL implementations. Ignoring the implementa-
tion detail is simply following the underlying principle of using abstract data
types. It is precisely the implementation details that make testing of STL code
difficult: an incorrect use of the STL may, in fact, work in a particular implementa-
tion of the STL. However, this “correct” behavior will be both non-portable
and likely to break if changes are made to the code, such as the ordering of
structures in memory. This difficult-to-test, difficult-to-reproduce behavior, typi-
cal of pointer and memory errors, makes a strong case for verification that code
relies only on behavior guaranteed by the Standard Template Library’s definition
in the C++ language standard [9], which is precisely what we provide.

Our approach is to produce an operational model of the behavior guaran-
teed by the STL standard and apply predicate abstraction to a modified C++
program in which STL calls have been replaced by the operationally equivalent
model. In particular, our verification tool is a predicate abstraction-based model
checker that handles a large subset of the C++ language, and our operational
model is written in (a variation of) C++. The operational model is not an imple-
mentation of the Standard Template Library, as it makes use of non-executable
features such as infinite arrays — supported by the logic of our model checker,
but not realizable in compiled code. The C++ model checker handles STL code,
once it has been rewritten using the operational model, with the same standard
abstraction-refinement loop as is used for the rest of the program. We show that
it suffices to verify correctness using the operational model by proving that the
preconditions on operations in the model imply the preconditions guaranteed by
the language definition for those operations, and the post-conditions given by
the standard imply the strongest post-conditions for the operational model.

The contribution of this work is to extend the powerful predicate abstrac-
tion technique for software model checking to apply to C++ programs, and in
particular to use an operational model and the principles of abstract data types
to efficiently verify usage of the C++ Standard Template Library [7, 9] in an
implementation-independent manner.
Related Work  Our approach to model checking code calling the Standard Template Library is based on a variation of predicate abstraction [6], and inspired by the recent success of software model checkers [10, 4, 1, 5] based on predicate abstraction and counterexample-guided abstraction refinement [11].

We extend our SAT-based predicate abstraction [5] to handle a large subset of the C++ language, including objects, (operator) overloading, references and templates (without specialization). Previous abstraction-based model checkers neither handle C++ programs nor provide an operational semantics supporting implementation-independent verification of code using the STL.

Wang and Musser’s effort to verify template code in C++ is a dynamic approach, based on gdb; it is capable of providing correctness proofs only if loop invariants are provided by a programmer [12]. The CMC model checker [13] can, in theory, verify C++ code compiled with templates and STL constructs, but checks implementation-dependent behavior as it is an explicit-state exploration, actually executing the code (with the attendant scaling and completeness problems). Similarly, NASA’s JAVA PathFinder 2 [14] has been applied to Java code that makes use of standard Java class library containers, but also relies on explicit exploration, rather than an abstraction capturing the guaranteed behaviors of the abstract data types.

Outline  Section 2 presents a Hoare-style formalization of the semantics of the STL as described in the C++ standard. Section 3 describes the operational model. We conclude with experimental results in Section 4.

2 Axiomatic Semantics

The C++ standard defines the semantics of the STL informally using pre- and post-conditions. We axiomatically formalize the semantics of the standard sequential containers list, vector, deque. The semantics of associative containers such as map, multimap, set and multiset is defined using a similar way. Therefore we omit their presentation. We define Hoare triples in the "forward"-style for the methods of the container classes. "Backward"-style axioms for the purpose of generating verification conditions can be derived using the consequence rule. Hoare-style axiomatizations of languages that permit aliasing are problematic [15–18]; we reduce the aliasing problem between iterators to aliasing between elements of an array.

The constructors of the containers have trivial semantics (either creating an empty container, or copying an existing container). We omit their axiomatizations. Methods such as push_back() and pop_front() are syntactic sugar for insert() and erase(). We therefore limit the presentation to insert() and erase(). Furthermore, the standard defines several forms of insert() and erase() methods. Since they can be implemented with the help of each other, we present only one of each category.
2.1 The Assertion Language

We distinguish three types of variables: we define the set of container variables $\mathcal{C}$, the set of integer variables $\mathcal{N}$, and the iterator variables $\mathcal{I}$. The set of variables is denoted by $\mathcal{V} = \mathcal{C} \cup \mathcal{I} \cup \mathcal{N}$. By convention, we assume $\{c, d, l, m, v\} \subset \mathcal{C}, \{i, j, n\} \subset \mathcal{N}$, and $\{it, it_1, it_2\} \subset \mathcal{I}$. We assume that the containers contain elements of some type $\mathcal{T}$. We denote the set of variables of this type by $\mathcal{T}$, and by convention, $t \in \mathcal{T}$.

We distinguish two different kind of container variables: active and inactive containers. By convention, we denote active containers with unprimed variables, e.g., $c, v, d$, and inactive containers by primed variables, e.g., $c', v', d'$. Inactive container variables are used in post-conditions to denote the pre-state of containers. The set of active containers is denoted by $\mathcal{A} \subset \mathcal{C}$.

We define the syntax for integer expressions (IntExpr) in the usual manner:

$$
\text{IntExpr} := \mathcal{N} \mid \mathcal{Z} \\
\quad \mid \mathcal{C}.\text{size} \mid \mathcal{C}.\text{capa} \\
\quad \mid \text{IntExpr} ( + \mid - \mid \ast \mid \ldots ) \text{IntExpr}
$$

The expressions $c.\text{size}$ and $c.\text{capa}$ denote the size and the capacity of a container $c$, respectively. We define the following iterator expressions:

$$
\text{ItExpr} := \mathcal{I} \mid \text{ItExpr} ( + \mid - ) \text{IntExpr} \\
\quad \mid \mathcal{C}.\text{begin}() \mid \mathcal{C}.\text{end}()
$$

Note that expressions of iterator type used in the program may contain additional operators, e.g., the dereferencing operator defined below. These operators are not permitted in assertions. We define the following expressions of type $\mathcal{T}$:

$$
\text{TExpr} := \mathcal{T} \mid \mathcal{C}_{\text{IntExpr}}
$$

The expression $c_i$ denotes the value of the $i^{th}$ element of the container $c$.

Note that in order to avoid some substitution details in the following rules, we assume the expressions in commands to be variable expressions.

Assertions may relate integers, compare container elements and iterators, relate iterators to container elements, and may contain the usual Boolean connectives:

$$
\text{Assert} := \text{IntExpr} ( < \mid = \mid \ldots ) \text{IntExpr} \\
\quad \mid \text{TExpr} = \text{TExpr} \mid \text{ItExpr} = \text{ItExpr} \\
\quad \mid \text{ItExpr} \; \mapsto \; \mathcal{C} \\
\quad \mid \neg \text{Assert} \mid \text{Assert} ( \lor \mid \land \mid \ldots ) \text{Assert} \\
\quad \mid \forall \text{var} \cdot \text{Assert} \mid \exists \text{var} \cdot \text{Assert}
$$

By $it \mapsto c$ we denote the fact that the iterator $it$ points to the $i^{th}$ element of the container $c$. As a special case, $i$ may be equal to the number of elements in the container. In this case, we say that $i$ points to the end of the container $c$. The operator $i \mapsto c$ is only defined for offsets $i \in \{0, \ldots, c.\text{size}\}$.
2.2 Iterators

We first formalize the concept of the Iterator, which is technically a pointer to an element inside of a container. Besides iterators, the C++ standard permits references to the elements inside a container. For all containers except `deque<T>`, references can be replaced trivially by iterators. We postpone the discussion how references to elements inside a `deque` are handled.

Figure 1 shows the axiomatization of the semantics of the operations on iterators. Iterators are typically created using the `begin()` and `end()` methods of containers. This is axiomatized by the two schemata `it-begin` and `it-end`.

Two iterators that point to the same location are equal (schema `it-eq`). To argue that two iterators are not equal it is neccessary to show that they point to two different positions inside the same container (schema `it-neq`).

All containers permit incrementing and decrementing an iterator. If `it` points to the position `i` inside container `c`, then `it + 1` points to the position `i + 1` (schema `it-inc`). Note that `it + 1` may be `c.end()`. Similarly, if `it` points to the position `i` and `i` is greater than zero, then `it − 1` points to the position `i − 1` (schema `it-deq`).

In addition to the previous axiom schemata, we provide the semantics of the mutation of iterators, the dereferencing commands, and of the distance between two iterators in Fig. 2.

2.3 Sequential Containers

The sequential containers `list`, `vector` and `deque` conform to a common basic semantics described by the rules `seq-ins` and `seq-era` given in Figure 3.

Let `c` denote an instance of a sequential container with elements of type `T`. The `insert()` method takes an iterator `it_1` and a reference to an object of type `T` as arguments. As a pre-condition, `it_1` must point to an element in `c` or be equal to `c.end()`. The post-condition guarantees that `it_2` points to the newly inserted element.

The `erase()` method removes the element pointed to by the iterator `it_1` from the container `c`. The post-condition guarantees that the iterator `it_2` points to the position in the sequence that was just beyond the erased element. The post-conditions of `insert()` and `erase()` for the validity of iterators depend on the particular container type, and are formalized in the following.
\[
\begin{align*}
\{ P \land it \xrightarrow{i} c \land 0 \leq i + j \leq c.\text{size} \} & \quad \text{it} := j & \quad \text{(it-mut-inc)}
\{ P[\text{it}/\text{it}'] \land \text{it} \xrightarrow{i+j} c \} & \quad c \in \mathcal{A}
\{ P \land it \xrightarrow{i} c \land i < c.\text{size} \} & \quad \text{it} := t & \quad \text{(it-deref1)}
\{ P[c/c'] \land c_i = t \land \\
\forall j \neq i. c_j = c'_j \} & \quad c \in \mathcal{A}
\end{align*}
\]

\[
\begin{align*}
\{ P \land it \xrightarrow{i} c \land i < c.\text{size} \} & \quad t := \star \text{it} & \quad \text{(it-deref2)}
\{ P[\text{it}']/\text{it} \} & \quad c \in \mathcal{A}
\{ P \land it_2 \xrightarrow{j} c \land it_1 \xrightarrow{i} c \} & \quad n := it_2 - it_1 & \quad \text{(it-dist)}
\{ P[\text{n}/\text{n}'] \land n = j - i \} & \quad c \in \mathcal{A}
\end{align*}
\]

Fig. 2. Rules for iterator

\[
\begin{align*}
\{ P \land it_1 \xrightarrow{i} c \} & \quad it_2 := c.\text{insert}(it_1, t) & \quad \text{(seq-ins)}
\{ P[c/c'] \land i' = i[c/c'] \land \\
\quad it_2 \xrightarrow{i'} c \land c.\text{size} = c'.\text{size} + 1 \land c_{i'} = t \land \\
\quad \forall j < i'. c_j = c'_j \land \\
\quad \forall j \geq i'. c_j + 1 = c'_j \} & \quad it_2 := c.\text{erase}(it_1) & \quad \text{(seq-era)}
\end{align*}
\]

Fig. 3. Basic Rules for Sequential Containers

The list Container Figure 4 shows the additional rules for lists. Let \( l \) be an active instance of list<T>. The insert() method takes an iterator \( it_1 \) and a reference to an object of type T. As a pre-condition, \( it_1 \) must point to an element of \( l \) or be equal to \( l.\text{end}() \). The post-condition guarantees that an iterator valid in the pre-state is also valid in the post-state.

The erase() method removes the element pointed to by the iterator \( it_1 \) from the list \( l \). The post-condition provides no guarantee about the validity of the iterators that were pointing to the erased element, but any other iterators are not affected by the removal. Note that the post-condition does not guarantee that iterators to the erased element are invalid; however, previous guarantees about the validity of such iterators cannot carry over from the pre-condition, as the container is renamed. Thus, no conclusions can be made about the validity or invalidity of such iterators in the post-state.
The vector Container Figure 5 shows the additional rules for vectors. Let \( v \) be an instance of \texttt{vector\{T\}}. A vector \( v \) has a capacity \( v.capa \) that corresponds to the number of elements \( v \) can hold without having to reallocate its content. Therefore, there are two different rules for the \texttt{insert()} method. If no reallocation occurs (schema \texttt{vec-ins1}), the post-condition guarantees that the iterators pointing before the inserted element are still valid. Otherwise (schema \texttt{vec-ins2}), no guarantee is provided in the post-state about the validity of the iterators that were pointing to or beyond the erased element. The validity of other iterators is not affected by the removal.

The \texttt{erase()} method removes the element contained in the vector \( v \) and pointed to by the iterator \( it_1 \). The post-condition does not provide any guarantees about the validity of the iterators that were pointing to or beyond the erased element. The validity of other iterators is not affected by the removal.
The `reserve()` method adjusts the capacity of the vector. After its invocation, the capacity of the vector is greater or equal to the argument \( n \). If the capacity in the pre-state is less than \( n \), a reallocation occurs and the capacity is increased. Otherwise, the invocation has no effect.

The **deque Container** The deque is a container for which insert and erase operations at either end of the sequence are optimized. The deque differs from other containers. The validity of the iterators and references to the elements in the sequence do not follow the same policy. For instance, an insert at either end of a deque invalidates all the iterators but has no effect on the references. Therefore, we have to distinguish references from iterators.

The rule `dq-e-inv` describes the effects of an insertion at either end of a deque. Let \( d \) be an instance of `deque<T>`. The pre-condition asserts that the iterator \( it_1 \) points either to the first element of \( d \) or to its end. The post-condition guarantees that the validity of the references \( ref \) do not change. It provides no guarantee about the validity of the iterators of \( d \). An insertion in the middle of a deque invalidates both the references and the iterators. Thus, there is no specific rule for this case.

The `erase()` method removes the element pointed to by the iterator \( it_1 \) from the deque \( d \). If the element is at either end of \( d \), then the operation has no effect on the validity of the iterators and references that were not pointing to the erased element (rule `dq-e-era`). A removal of an element in the middle of a deque invalidates both the references and the iterators.

The **map Container** A `map<K, T, < >` associates unique keys of type \( K \) to values of type \( T \). The template argument \( < \) is a predicate function that must induce a strict weak ordering relation on the elements of \( K \). The equivalence of
\[
\begin{align*}
&\{ P \land \forall i. m_i \text{. first } \equiv t \text{. first} \} p := m \text{. insert}(t) \\
&\{ P[m/m'][p/p'] \land \\
&\exists i \leq m' \text{. size } . \\
&\forall j < i. m_j \text{. first } < t \text{. first} \\
&\forall j \geq i. t \text{. first } < m_j \text{. first} \\
&p \text{. first } m \land p \text{. second } = \text{true} \land m_i = t \land m \text{. size } = m' \text{. size } + 1 \\
&\forall i, it, j < i. it \downarrow j \to m' \Rightarrow it \downarrow j \to m \\
&\forall i, it, j \geq i. it \downarrow j \to m' \Rightarrow it \downarrow j+1 \to m \\
&\forall j < i. m_j = m_j' \\
&\forall j \geq i. m_{j+1} = m_j' \\
&\} \\
\end{align*}
\]

\[
\begin{align*}
&\{ P \land m_i \text{. first } \equiv t \text{. first} \} p := m \text{. insert}(t) \\
&\{ P[p/p'] \land p \text{. second } = \text{false} \}
\end{align*}
\]

\[
\begin{align*}
&\{ P \land it \downarrow i \to m \land i < m \text{. size} \} m \text{. erase}(it) \\
&\{ P[m/m'] \land i' = i[m/m'] \land m \text{. size } = m' \text{. size } - 1 \land \\
&\forall i, it_2, j < i'. it_2 \downarrow j \to m' \Rightarrow it_2 \downarrow j \to m \\
&\forall i, it_2, j \geq i'. it_2 \downarrow j-1 \to m' \Rightarrow it_2 \downarrow j \to m \\
&\forall j < i. m_j = m_j' \\
&\forall j \geq i. m_{j-1} = m_j' \\
&\} \\
\end{align*}
\]

\[
\begin{align*}
&\{ P \land m_i \text{. first } \equiv k \} it := m \text{. find}(k)\{ P[it/it'] \land it \downarrow i \to m \}
\end{align*}
\]

\[
\begin{align*}
&\{ P \land \forall i. m_i \text{. first } \equiv k \} it := m \text{. find}(k)\{ P[it/it'] \land it \downarrow m \}
\end{align*}
\]

Fig. 7. Rules for map

the keys noted \( \equiv \) is defined as follows:

\[
k_1 \equiv k_2 \Leftrightarrow \neg(k_1 < k_2) \land \neg(k_2 < k_1)
\]

The \( \text{insert()} \) method described here takes an argument \( t \in K \times T \) (Figure 7). The first component is denoted by \( t \text{. first} \) and corresponds to a key. The second component is denoted by \( t \text{. second} \) and corresponds to the value the key is mapped to. The tuple \( t \) is inserted into the map \( m \) if and only if no key is equivalent to \( t \text{. first} \). The returned value is a pair of an iterator and a Boolean value. Its second component is true if and only if \( t \) is inserted. In this case, the iterator \( p \text{. first} \) points to the newly inserted tuple \( t \). The post-condition guarantees that the validity of the iterators is not affected by the insertion.

The \( \text{erase()} \) method removes the tuple pointed to by the argument \( it \) from the map. The post-condition does not provide any guarantees about the new value of the iterators that were pointing to the erased element. The validity of other iterators is not affected by the removal.

The \( \text{erase()} \) and \( \text{insert()} \) methods of the \texttt{multimap}, \texttt{set} and \texttt{multiset} containers behave in the same way as the ones of \texttt{map}: the insertion of an element
does not invalidate iterators; erasing of an element only invalidates the iterators pointing to that element.

The `find()` method searches for a tuple of map $m$ with a key equivalent to the argument $k$. If such a tuple exists, then the returned value is an iterator pointing to it. Otherwise, the returned iterator points to the end of the map. Since the definition of the semantics of other methods such as `lower_bound()`, `upper_bound()`, and `equal_range()` follows a similar pattern, we skip their presentation.

3 An Operational Model for the STL

In order to verify that a program using the STL obeys the pre-conditions of the methods of the containers and iterators as formalized above, we use an operational model. The operational model assumes that variables with an array type of infinite size can be declared, i.e., mappings from $\mathbb{N}_0$ into some arbitrary domain. Note that the operational model is therefore optimized for verification purposes, and is not actually executable.

The model is expressed using the Hoare Logic style. In the following, we use $X$ and $Y$ as meta types. Let $It$ and $Cont$ denote respectively the set of iterators and container values. The set of variables of a specific type $X$ is written $V_X$. The set of states is denoted by $\Sigma$. A state $s$ is a tuple of functions from variables to values. The symbol $\llbracket X \rrbracket$ denotes a function from states and expressions to values of type $X$. The symbol $\llbracket cmd \rrbracket$ denotes a function from states and expressions to states. The definitions of the previous sets and functions are shown in Fig. 8. Furthermore, note that containers have a field `capa`, which is only used if the container is a vector.

We relate the sets of the axiomatic model and the sets of the operational model in the following way: $V_{It} \subset \mathcal{I}$, $V_{Cont} = \mathcal{A}$, $V_T \subset \mathcal{T}$ and $V_Z \subset \mathcal{N}$.

Let $x$ denote a variable, $e$ an expression and $c$ a container variable. Fig. 9 shows the meaning of some of the expressions of the language. The semantics of the trivial expressions are skipped. For the sake of conciseness, the language used for the operational model has new constructs such as the ones found in
expr-var, expr-vcont or expr-func. Note that in expr-vcont, \( \hat{c} \) denotes the variable \( c \) itself and not its value.

A version number is associated with each offset of the data array of a container. The version and data arrays can be seen as functions \( \mathbb{N}_0 \) to respectively \( \mathbb{N}_0 \) and \( T \). Each iterator has a field called version, which is a number. The field \( \hat{v}_\text{cont} \in V_{\text{Cont}} \cup \{\bot\} \) identifies into the container into which an iterator points, or is \( \bot \) in the case of an iterator that has not yet been assigned to.

Our operational model maintains the following invariant: An iterator \( \text{it} \) points into a container \( c \) if and only if the version of the iterator matches the version of the element it points to:

\[
\begin{align*}
\models \text{it} \mapsto c & \iff \models \text{it}.v\text{cont} = \hat{c} \land \text{it}.\text{offset} = i \land \\
\text{it}.\text{version} & = c.\text{version}(i)
\end{align*}
\]

We use \( s(a, x) \) to denote the state equal to \( s \) except that the value of the variable \( a \) is \( x \). If \( a \) has a field named \( b \), then \( s(a.b, x) \) denotes the state that is equal to \( s \) except that \( a.b \) is \( x \). For arrays, we use the notation \( s(c_i, t) \) to refer to the state equal to \( s \) except that the \( i \)th element of \( c \) is equal to \( t \). For convenience, we use \( s[..|a_i, x_i|..] \) to denote the state obtained from \( s \) by simultaneously substituting all \( a_i \) by \( x_i \).

We translate the axiomatic semantics of the iterators into an operational model (Fig. 10). Note that \( \text{I} \) and \( \text{J} \) denote macros used for shortening the formulas.

**The Operational Semantics of Vectors** We present the operational semantics of the insertion and the removal of an element of a vector in Fig. 11. The rule \( \text{opm-lst-ins} \) describes the operational semantics of the command \( \text{it}_2 := v.\text{insert}(\text{it}_1, t) \); by means of the program \( I_{\text{vec}} \) given in Fig. 12. The program \( I_{\text{vec}} \) inserts the value \( t \) into the vector \( c \) just before the position pointed to by the iterator \( \text{it}_1 \). The iterator \( \text{it}_2 \) is then set to the newly inserted element. The validity of the iterators depends on the capacity of the vector.

The rule \( \text{opm-vec-era} \) describes the effect of removing an element form a vector by means of the program \( E_{\text{vec}} \), given in Fig. 13. Note that only the iterators that point beyond the erased element are invalidated.

**The Operational Semantics of a List** Fig. 14 shows a program that inserts a value into a list. Note that due to the universal quantifier in the post-condition.
of the rule \textit{bst-ins}, every iterator variable may need to be updated. In order to overcome the issues that arise with the use of universal quantifiers we propose an over-approximation. Note that we require the over-approximation to be sound, i.e., the checker does not incorrectly report that a program is correct.

One possible over-approximation for a list consists in keeping valid only the iterators whose offsets are not affected. The checker may as a result report spurious counterexamples, but the approximation may be sufficient for proving the correctness of some properties. Fig. 15 shows the insertion of an element into a list, using the over-approximation just mentioned.

The translation of the remaining axiomatic rules for STL into our operational semantics can be carried out in the same manner. We therefore omit their presentation. The translation of the operational model into a C++ library that is used for model checking an application is straightforward, though an over-approximation is necessary to handle the quantifiers. As a possible improvement of the model checker, one can think of implementing a loop refinement procedure for ruling out the spurious counter examples.

Depending on the property being checked, it may even be sufficient to adopt a coarser over-approximation that has the benefit of making the verification more efficient. Instead of considering an array of version numbers, one can associate
1: procedure insert_vec
2: \begin{align*}
& v\.size := v\.size + 1; \\
& v\.data := \lambda i. \begin{cases} 
& 0 < i & v\.data(i) \\
& i = \it_1\.offset & t : v\.data(i-1) \\
& \text{else} 
\end{cases}; \\
& \text{if } v\.size \leq v.capa \text{ then} \\
& \quad v\.version := \lambda i. \begin{cases} 
& 0 < i & v\.version(i) \\
& \text{else} 
\end{cases}; \\
& \text{else} \\
& \quad v\.version := \lambda i. v\.version(i)+1; \\
& \quad NonDetVal \text{ stands for a non-deterministic strictly positive value} \\
& \quad v\.capa := v\.capa + NonDetVal; \\
& \it_2 := (\hat{v}, \it_1\.offset, l\.version(\it_1\.offset));
\end{align*}

Fig. 12. The program $I_{vec}$ inserts into the vector $v$ the element $t$.

1: procedure erase_vec
2: \begin{align*}
& v\.size := v\.size - 1; \\
& v\.version := \lambda i. \begin{cases} 
& 0 < i & v\.version(i) \\
& \text{else} 
\end{cases} + 1; \\
& v\.data := \lambda i. \begin{cases} 
& 0 < i & v\.data(i) \\
& \text{else} 
\end{cases} + 1; \\
& \it_2 := (\hat{v}, \it_1\.offset, v\.version(\it_1\.offset));
\end{align*}

Fig. 13. The program $E_{vec}$ removes from the vector $v$ the element pointed to by $\it_1$.

a single version number with a whole container. Every time the version of a container is increased, all the iterators pointing to it are invalidated.

4 Experimental Results

Our implementation is based on SatAbs, which uses a SAT-solver to compute the abstract model [5]. The operational model uses an unbounded array in order to store the container elements. We therefore extend SatAbs in order to support unbounded arrays in the predicates and $R$. We first reduce the formula with array operations to a formula over uninterpreted functions. This formula is then reduced to bit-vector logic by means of Ackermann’s reduction. This is an eager reduction, and is therefore similar to the implementation in UCLID [19].

Our front-end to SatAbs supports a large subset of the C++ language. We currently lack support for friend member functions, template specialization, virtual functions, and virtual inheritance.

We use MiniSAT as benchmark for our technique. MiniSAT is “a minimalistic, open-source SAT solver,” recognized in the SAT 2005 competition as one of the most efficient SAT solvers available [20]. The importance of effective SAT solvers to many applications, particularly verification, is well known, and MiniSAT is a popular base for cutting-edge research in Boolean satisfiability.

A number of variants of MiniSAT are available. The standard release is written in C++. One of these variants replaces the custom made dynamic vector used in the main releases with the vector class provided by the C++ Standard
1: **procedure insert_lst1**
2:  \( l\).size := \( l\).size + 1 ;
3:  \( l\).data := \( \lambda \) i. i < \( it_1\).offset \? i = \( it_1\).offset \? t : \( l\).data(i-1);
4:  **for all** var \( \in \ V_{\{it_1, it_2\}} \) **do**
5:  \( \text{var}.offset := \text{var}.offset < \text{it}_1.\text{offset} \? \text{var}.\text{offset} : \text{var}.\text{offset}+1;\)
6:  \( \text{var}.\text{version} := l.\text{version}(\text{var}.\text{offset});\)
7:  \( it_2 := (l, it_1.\text{offset}, l.\text{version}(it_1.\text{offset}));\)
8:  \( it_1.\text{offset} := it_1.\text{offset} + 1;\)
9:  \( it_1.\text{version} := l.\text{version}(it_1.\text{offset});\)

**Fig. 14.** The program \( I_{\text{lst}} \) inserted in the list \( l \) the element \( t \) before the position pointed to by iterator \( it_1 \).

1: **procedure insert_lst2**
2:  \( l\).size := \( l\).size + 1 ;
3:  \( l\).data := \( \lambda \) i. i < \( it_1\).offset \? \( l\).data(i) : i = \( it_1\).offset \? t : \( l\).data(i-1) ;
4:  \( l\).version := \( \lambda \) i. i < \( it_1\).offset \? \( l\).version(i) : \( l\).version(i)+1;
5:  \( it_2 := (l, it_1.\text{offset}, l.\text{version}(\text{offset}));\)

**Fig. 15.** The program \( I_{\text{lst2}} \) inserts into the list \( l \) the element \( t \). Note this is an over-approximation

Template Library. The MiniSAT code is hand-crafted for high performance, and makes use of templates, references, and operator overloading.

We obtain a total of 139 non-trivial verification conditions for the MiniSAT code, out of which 115 are due to the pre-conditions of our operational version of the vector class. We use a limit of 50 refinement iterations. The benchmarks were performed on a Linux machine with a 2.8 GHz Intel Xenon processor.

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Avg. Time(min)</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failed</td>
<td>33</td>
<td>59.9</td>
<td>37</td>
<td>112</td>
</tr>
<tr>
<td>Success</td>
<td>103</td>
<td>18.8</td>
<td>&lt;1</td>
<td>85</td>
</tr>
<tr>
<td>Counterexample</td>
<td>3</td>
<td>13.7</td>
<td>6</td>
<td>18</td>
</tr>
</tbody>
</table>

We were able to prove 103 properties (74%) in an average of 19 minutes each, and obtained counterexamples for 3 properties. The counterexamples are due to imprecise modeling of the environment. As an example, MiniSAT contains an assertion that compares an integer read from a file with a constant. For 33 properties, the iteration limit was exceeded. The model checker and the operational model of STL are available to other researchers for experimentation \(^3\).

### 5 Conclusion

We have shown how an operational semantics for the defined behavior of the C++ Standard Template Library may be used to verify programs with STL

\(^3\) [http://www.inf.ethz.ch/personal/daniekro/satabs/](http://www.inf.ethz.ch/personal/daniekro/satabs/)
data structures in an implementation-independent manner, leveraging the high-
level nature of abstract data types to aid predicate abstraction. This approach
relies on the first reported symbolic model checking for complex C++ code,
implemented in the SatAbs model checker. Experimental results show the utility
of this approach in finding errors and verifying correct code in realistic software
programs, including tools used in formal verification.

References

1. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular verification of software
2. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog
of Programming Languages. (2002) 58–70
X3J16/94-0095, ISO WG21/N0482 (1994)
Symposium on very high level languages, ACM (1974) 50–59
10. Ball, T., Rajamani, S.: Automatically validating temporal safety properties of
proach to model checking real code. In: Symposium on Operating System Design
and Implementation. (2002)
15. Hoare, C.A.R., Wirth, N.: An axiomatic definition of the programming language
pascal. Acta Informatica 2 (1973)
Construction. (2000) 102–126
17. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In:
LICS, IEEE (2002) 55–74
18. Ishtiaq, S.S., O’Hearn, P.W.: BI as an assertion language for mutable data struc-
logic of counter arithmetic with lambda expressions and uninterpreted functions.