Verifying C++ with STL Containers via Predicate Abstraction*

Nicolas Blanc[1], Daniel Kroening[1], and Alex Groce[2]

[1] Computer Systems Institute, ETH Zürich
[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" 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

* 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 implementation detail is simply following the underlying principle of using abstact 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 implementation 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, typical 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 guaranteed 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 implementation 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 abstraction 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. Therfore 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 $\mathcal{V} = \mathcal{C} \dot{\cup} \mathcal{I} \dot{\cup} \mathcal{N}$. 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 `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:

$$\begin{aligned} \mathit{IntExpr} := {} & \mathcal{N} \mid \mathbb{Z} \\ & \mid \mathcal{C}.\mathbf{size} \mid \mathcal{C}.\mathbf{capa} \\ & \mid \mathit{IntExpr}\ (+ \mid - \mid * \mid ...)\ \mathit{IntExpr} \end{aligned}$$

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

$$\begin{aligned} \mathit{ItExpr} := {} & \mathcal{I} \mid \mathit{ItExpr}\ (+ \mid -)\ \mathit{IntExpr} \\ & \mid \mathcal{C}.\mathbf{begin}() \mid \mathcal{C}.\mathbf{end}() \end{aligned}$$

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 `T`:

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

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:

$$\begin{aligned} \mathit{Assert} := {} & \mathit{IntExpr}\ (< \mid = \mid ...)\ \mathit{IntExpr} \\ & \mid \mathit{TExpr} = \mathit{TExpr} \mid \mathit{ItExpr} = \mathit{ItExpr} \\ & \mid \mathit{ItExpr} \overset{\mathit{IntExpr}}{\rightarrowtail} \mathcal{C} \\ & \mid \neg \mathit{Assert} \mid \mathit{Assert}\ (\vee \mid \wedge \mid ...)\ \mathit{Assert} \\ & \mid \forall \mathit{var}.\ \mathit{Assert} \mid \exists \mathit{var}.\ \mathit{Assert} \end{aligned}$$

By $it \overset{i}{\rightarrowtail} c$ 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 $\overset{i}{\rightarrowtail} c$ is only defined for offsets $i \in \{0, \ldots, c.size\}$.

$$\frac{}{c.begin() \overset{0}{\rightarrowtail} c} \text{ (it-begin)} \qquad \frac{}{c.end() \overset{c.size}{\rightarrowtail} c} \text{ (it-end)}$$

$$\frac{it_1 \overset{i}{\rightarrowtail} c \wedge it_2 \overset{i}{\rightarrowtail} c}{it_1 = it_2} \text{ (it-eq)} \qquad \frac{it_1 \overset{i}{\rightarrowtail} c \wedge it_2 \overset{j}{\rightarrowtail} c \wedge i \neq j}{it_1 \neq it_2} \text{ (it-neq)}$$

$$\frac{it \overset{i}{\rightarrowtail} c \wedge i < c.size}{it + 1 \overset{i+1}{\rightarrowtail} c} \text{ (it-inc)} \qquad \frac{it \overset{i}{\rightarrowtail} c \wedge 0 < i}{it - 1 \overset{i-1}{\rightarrowtail} c} \text{ (it-dec)}$$

Fig. 1. Axiomatization of Iterators

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.

$$\{\ P \wedge it \overset{i}{\rightarrowtail} c \wedge 0 \leq i + j \leq c.size\ \}\ it{+}{=}j \qquad \text{(it-mut-inc)}$$
$$\{\ P[it/it'] \wedge it \overset{i+j}{\rightarrowtail} c\ \} \qquad c \in \mathcal{A}$$

$$\{\ P \wedge it \overset{i}{\rightarrowtail} c \wedge i < c.size\ \} \qquad *it := t \qquad \text{(it-deref1)}$$
$$\{\ P[c/c'] \wedge c_i = t \wedge \quad \forall j \neq i\,.\,c_j = c'_j \wedge\} \qquad c \in \mathcal{A}$$

$$\{\ P \wedge it \overset{i}{\rightarrowtail} c \wedge i < c.size\ \} \qquad t := *it \qquad \text{(it-deref2)}$$
$$\{\ P[t/t'] \wedge t = c_i\ \} \qquad c \in \mathcal{A}$$

$$\{\ P \wedge it_2 \overset{j}{\rightarrowtail} c \wedge it_1 \overset{i}{\rightarrowtail} c\ \} \qquad n := it_2 - it_1 \qquad \text{(it-dist)}$$
$$\{\ P[n/n'] \wedge n = j - i\ \} \qquad c \in \mathcal{A}$$

Fig. 2. Rules for *iterator*

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} c\}\ it_2 := c.insert(it_1, t) \qquad \text{(seq-ins)}$$
$$\{\ P[c/c'][it_2/it'_2] \wedge i' = i[c/c'] \wedge \quad it_2 \overset{i'}{\rightarrowtail} c \wedge c.size = c'.size + 1 \wedge c_{i'} = t \wedge \quad \forall j < i'\,.\,c_j = c'_j \wedge \quad \forall j \geq i'\,.\,c_{j+1} = c'_j\ \}$$

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} c \wedge i < c.size\}\ it_2 := c.erase(it_1) \qquad \text{(seq-era)}$$
$$\{\ P[c/c'][it_2/it'_2] \wedge i' = i[c/c'] \wedge \quad it_2 \overset{i'}{\rightarrowtail} c \wedge c.size = c'.size - 1 \wedge \quad \forall j < i'\,.\,c_j = c'_j \wedge \quad \forall j > i'\,.\,c_{j-1} = c'_j\ \}$$

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 $l.end()$. 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.

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} l\}\ it_2 := l.insert(it_1, t) \qquad \text{(lst-ins)}$$
$$\{\ P[l/l'][it_2/it_2'] \wedge i' = i[l/l'] \wedge$$
$$\forall it, j < i' \,.\, it \overset{j}{\rightarrowtail} l' \Rightarrow it \overset{j}{\rightarrowtail} l \wedge$$
$$\forall it, j \geq i' \,.\, it \overset{j}{\rightarrowtail} l' \Rightarrow it \overset{j+1}{\rightarrowtail} l\ \}$$

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} l \wedge i < l.size\}\ it_2 := l.erase(it_1) \qquad \text{(lst-era)}$$
$$\{\ P[l/l'][it_2/it_2'] \wedge i' = i[l/l'] \wedge$$
$$\forall it, j < i' \,.\, it \overset{j}{\rightarrowtail} l' \Rightarrow it \overset{j}{\rightarrowtail} l \wedge$$
$$\forall it, j > i' \,.\, it \overset{j}{\rightarrowtail} l' \Rightarrow it \overset{j-1}{\rightarrowtail} l\ \}$$

Fig. 4. Additional Rules for `list`

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} v \wedge v.size < v.capa\}\ it_2 := v.insert(it_1, t) \qquad \text{(vec-ins1)}$$
$$\{\ P[v/v'][it_2/it_2'] \wedge i' = i[v/v'] \wedge v.capa = v'.capa \wedge$$
$$\forall it, j < i' \,.\, it \overset{j}{\rightarrowtail} v' \Rightarrow it \overset{j}{\rightarrowtail} v\}$$

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} v \wedge v.size = v.capa\}\ it_2 := v.insert(it_1, t) \qquad \text{(vec-ins2)}$$
$$\{\ P[v/v'][it_2/it_2'] \wedge v.capa \geq v.size\}$$

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} v \wedge i < v.size\}\ it_2 := v.erase(it_1) \qquad \text{(vec-era)}$$
$$\{\ P[v/v'][it_2/it_2'] \wedge i' = i[v/v'] \wedge v.capa = v'.capa \wedge$$
$$\forall it, j < i' \,.\, it \overset{j}{\rightarrowtail} v' \Rightarrow it \overset{j}{\rightarrowtail} v\}$$

$$\{\ P \wedge n \leq v.capa\}\ v.reserve(n)\{P\ \} \qquad \text{(vec-res1)}$$

$$\{\ P \wedge n > v.capa\}\ v.reserve(n) \qquad \text{(vec-res2)}$$
$$\{\ P[v/v'] \wedge v.size = v'.size \wedge v.capa \geq n \wedge \forall j \,.\, v_j = v_j'\ \}$$

Fig. 5. Additional Rules for `vector`

The `vector` Container Figure 5 shows the additional rules for vectors. Let v be an instance of `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 *insert*() method. If no reallocation occurs (schema *vec-ins1*), the post-condition guarantees that the iterators pointing before the inserted element are still valid. Otherwise (schema *vec-ins2*), no guarantee is provided in the post-state about the validity of the iterators that are pointing into v in the pre-state.

The *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.

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} d \wedge (i = d.size \vee i = 0)\}\ it_2 := d.insert(it_1, t) \qquad \text{(dqe-ins)}$$
$$\{\ P[d/d'][it_2/it_2'] \wedge i' = i[d/d'] \wedge$$
$$\forall ref, j < i' \,.\, ref \overset{j}{\rightarrowtail} d' \Rightarrow ref \overset{j}{\rightarrowtail} d \wedge$$
$$\forall ref, j \geq i' \,.\, ref \overset{j}{\rightarrowtail} d' \Rightarrow ref \overset{j+1}{\rightarrowtail} d\}$$

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} d \wedge (i = d.size - 1 \vee i = 0)\}\ it_2 := d.erase(it_1) \qquad \text{(dqe-era)}$$
$$\{\ P[d/d'][it_2/it_2'] \wedge i' = i[d/d'] \wedge$$
$$\forall it, j < i' \,.\, it \overset{j}{\rightarrowtail} d' \Rightarrow it \overset{j}{\rightarrowtail} d \wedge$$
$$\forall it, j > i' \,.\, it \overset{j}{\rightarrowtail} d' \Rightarrow it \overset{j-1}{\rightarrowtail} d \wedge$$
$$\forall ref, j < i' \,.\, ref \overset{j}{\rightarrowtail} d' \Rightarrow ref \overset{j}{\rightarrowtail} d \wedge$$
$$\forall ref, j > i' \,.\, ref \overset{j}{\rightarrowtail} d' \Rightarrow ref \overset{j-1}{\rightarrowtail} d\ \}$$

Fig. 6. Additional Rules for `deque`

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 *dqe-ins* 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 *dqe-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,` $\prec$ `>` associates unique keys of type K to values of type T. The template argument $\prec$ is a predicate function that must induce a strict weak ordering relation on the elements of K. The equivalence of

$$
\begin{array}{l}
\{\ P \wedge \forall i\,.\,\neg m_i.first \cong t.first\}\ p := m.insert(t) \\
\{\ P[m/m'][p/p'] \wedge \\
\quad \exists i \leq m'.size\,.\,\wedge \\
\quad \forall j < i\,.\,m'_j.first \prec t.first \wedge \\
\quad \forall j \geq i\,.\,t.first \prec m'_j.first \wedge \\
\quad p.first \overset{m}{\rightarrowtail} i \wedge p.second = true \wedge m_i = t \wedge m.size = m'.size + 1 \wedge \\
\quad \forall it, j < i\,.\,it \overset{j}{\rightarrowtail} m' \Rightarrow it \overset{j}{\rightarrowtail} m \wedge \\
\quad \forall it, j \geq i\,.\,it \overset{j}{\rightarrowtail} m' \Rightarrow it \overset{j+1}{\rightarrowtail} m \wedge \\
\quad \forall j < i\,.\,m_j = m'_j \wedge \\
\quad \forall j \geq i\,.\,m_{j+1} = m'_j\ \}
\end{array}
$$

$$
\begin{array}{l}
\{\ P \wedge m_i.first \cong t.first\}\ p := m.insert(t) \\
\{\ P[p/p'] \wedge p.second = false\ \}
\end{array}
$$

$$
\begin{array}{l}
\{\ P \wedge it \overset{i}{\rightarrowtail} m \wedge i < m.size\}\ m.erase(it) \\
\{\ P[m/m'] \wedge i' = i[m/m'] \wedge m.size = m'.size - 1 \wedge \\
\quad \forall it_2, j < i'\,.\,it_2 \overset{j}{\rightarrowtail} m' \Rightarrow it_2 \overset{j}{\rightarrowtail} m \wedge \\
\quad \forall it_2, j > i'\,.\,it_2 \overset{j-1}{\rightarrowtail} m' \Rightarrow it_2 \overset{j}{\rightarrowtail} m \wedge \\
\quad \forall j < i\,.\,m_j = m'_j \wedge \\
\quad \forall j > i\,.\,m_{j-1} = m'_j\ \}
\end{array}
$$

$$
\{\ P \wedge m_i.first \cong k\}\ it := m.find(k)\{P[it/it'] \wedge it \overset{i}{\rightarrowtail} m\ \}
$$

$$
\{\ P \wedge \forall i\,.\,\neg m_i.first \cong k\}\ it := m.find(k)\{P[it/it'] \wedge it \overset{m.size}{\rightarrowtail} m\ \}
$$

Fig. 7. Rules for `map`

the keys noted $\cong$ is defined as follows:

$$k_1 \cong k_2 \Leftrightarrow \neg(k_1 \prec k_2) \wedge \neg(k_2 \prec k_1)$$

The *insert*() method described here takes an argument $t \in K \times T$ (Figure 7). The first component is denoted by *t.first* and corresponds to a key. The second component is denoted by *t.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.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.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 *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 *erase*() and *insert*() methods of the `multimap`, `set` and `multiset` containers behave in the same way as the ones of 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 Σ. A state s is a tuple of functions from variables to values. The symbol $[\![]\!]_X$ denotes a function from states and expressions to values of type X. The symbol $[\![]\!]_{cmd}$ 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 $V_{\mathbb{Z}} \subset \mathcal{N}$.

$$
\begin{array}{lll}
It & = \{V_{Cont} \cup \bot\} \times \mathbb{N}_0 \times \mathbb{N}_0 & (\mathit{vcont}, \mathit{offset}, \mathit{version}) \quad \in It \\
Cont & = (\mathbb{N}_0 \to T) \times (\mathbb{N}_0 \to \mathbb{N}_0) & (\mathit{data}, \mathit{version}, \mathit{size}, \mathit{capa}) \in Cont \\
 & \quad \times \mathbb{N}_0 \times \mathbb{N}_0 & \\
\Sigma & = (V_Z \to \mathbb{Z}) \times (V_{It} \to It) & (\sigma_{\mathbb{Z}}, \sigma_{It}, \sigma_{Cont}, \sigma_T) \quad \in \Sigma \\
 & \quad \times (V_{Cont} \to Cont) \times (V_T \to T) & \\
[\![]\!]_X & : (X \ \mathit{Expression} \times \Sigma) \to X & \\
[\![]\!]_{cmd} & : (\mathit{Command} \times \Sigma) \to \Sigma &
\end{array}
$$

Fig. 8. The definitions of the functions and sets of the operational model.

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

$$
\begin{array}{lll}
[\![x]\!]_X s & = s.\sigma_X(x) & \text{(expr-var)} \\
[\![\mathring{c}]\!]_{V_{Cont}} s & = c & \text{(expr-vcont)} \\
[\![e_0 \,?\, e_1 \,:\, e_2]\!]_X s & = [\![e_0]\!]_{bool} s \,?\, [\![e_1]\!]_X s \,:\, [\![e_2]\!]_X s & \text{(expr-ite)} \\
[\![\lambda x.\, e]\!]_{X \to Y} s & = \lambda x'.\, [\![e]\!]_Y s\langle x,\, x'\rangle & \text{(expr-}\lambda\text{)} \\
[\![e_0(e_1)]\!]_Y s & = [\![e_0]\!]_{X \to Y} s([\![e_1]\!]_X s) & \text{(expr-func)} \\
[\![c_e]\!]_T s & = [\![c.data(e)]\!]_T & \text{(expr-at)}
\end{array}
$$

Fig. 9. The Operational Semantics of the expressions.

expr-ite, *expr-vcont* or *expr-func*. Note that in *expr-vcont*, $\mathring{c}$ 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 $vcont \in V_{Cont} \dot\cup \{\bot\}$ 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 it points into a container c if and only if the version of the iterator matches the version of the element it points to:

$$
s \vDash it \overset{i}{\rightarrowtail} c \iff s \vDash it.vcont = \mathring{c} \wedge it.offset = i \wedge \quad \text{(ass-ptsto)}
$$
$$
it.version = c.version(i)
$$

We use $s\langle a,\, x\rangle$ 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 $s\langle a.b,\, x\rangle$ denotes the state that is equal to s except that $a.b$ is x. For arrays, we use the notation $s\langle c_i,\, t\rangle$ to refer to the state equal to s except that the i^{th} element of c is equal to t. For convenience, we use $s\langle ..|a_i, x_i|..\rangle$ 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 **I** and **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 *opm-lst-ins* describes the operational semantics of the command $it_2 := v.insert(it_1, t)$; by means of the program I_{vec} given in Fig. 12. The program I_{vec} inserts the value t into the vector c just before the position pointed to by the iterator it_1. The iterator it_2 is then set to the newly inserted element. The validity of the iterators depends on the capacity of the vector.

The rule *opm-vec-era* describes the effect of removing an element form a vector by means of the program E_{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

$$\frac{s \vDash it \overset{\mathbf{I}}{\rightarrowtail} c \wedge 0 \leq \mathbf{I} + j < c.size}{s[\![it+=j]\!]_{cmd} s\langle \mathbf{I},\ [\![\mathbf{I}+j]\!]_{\mathbb{N}_0} s \mid it.version,\ [\![c.version(\mathbf{I}+j)]\!]_{\mathbb{N}_0} s\rangle} \quad \text{(opm-it-mut-inc)}$$
$$\mathbf{I} = it.offset$$

$$\frac{s \vDash it \overset{\mathbf{I}}{\rightarrowtail} c \wedge \mathbf{I} < c.size}{s[\![*it := t]\!]_{cmd} s\langle c_{[\![\mathbf{I}]\!]_{\mathbb{N}_0} s},\ [\![t]\!]_T s\rangle} \quad \text{(opm-it-deref1)}$$
$$\mathbf{I} = it.offset$$

$$\frac{s \vDash it \overset{\mathbf{I}}{\rightarrowtail} c \wedge \mathbf{I} < c.size}{s[\![t := *it]\!]_{cmd} s\langle t,\ [\![c_{\mathbf{I}}]\!]_T s\rangle} \quad \text{(opm-it-deref2)}$$
$$\mathbf{I} = it.offset$$

$$\frac{s \vDash it_2 \overset{it_2.offset}{\rightarrowtail} c \wedge it_1 \overset{it_1.offset}{\rightarrowtail} c}{s[\![i := it_2 - it_1]\!]_{cmd} s\langle i,\ it_2.offset - it_1.offset\rangle} \quad \text{(opm-it-dist)}$$

Fig. 10. The Operational Semantics of Iterators

$$\frac{s[\![I_{vec}]\!]_{cmd} s' \ \wedge\ s \vDash it_1 \overset{\mathbf{I}}{\rightarrowtail} v}{s[\![it_2 := v.insert(it_1, t)]\!]_{cmd} s'} \quad \text{(opm-vec-ins)}$$
$$\mathbf{I} = it_1.offset$$

$$\frac{s[\![E_{vec}]\!]_{cmd} s' \ \wedge\ s \vDash it_1 \overset{\mathbf{I}}{\rightarrowtail} v \wedge \mathbf{I} < v.size}{s[\![it_2 := l.erase(it_1)]\!]_{cmd} s'} \quad \text{(opm-vec-era)}$$
$$\mathbf{I} = it_1.offset$$

Fig. 11. The Operational Semantics of Vectors

of the rule *lst-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 straight-forward, 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:     v.size := v.size + 1;
3:     v.data := λ i. i < it1.offset ? v.data(i) : i = it1.offset ? t : v.data(i-1) ;
4:     if v.size ≤ v.capa then
5:         v.version := λ i. i < it1.offset ? v.version(i) : v.version(i)+1;
6:     else
7:         v.version := λ i. v.version(i)+1;
8:                 ▷ NonDetVal stands for a non-deterministic striclty positive value
9:         v.capa := v.capa + NonDetVal;
10:    it2 := (v̊, it1.offset, l.version(it1.offset) );
```

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

```
1: procedure erase_vec
2:     v.size := v.size - 1 ;
3:     v.version := λ i. i < it1.offset ? v.version(i) : v.version(i)+1;
4:     v.data := λ i. i < it1.offset ? v.data(i) : v.data(i+1);
5:     it2 := (v̊, it1.offset, v.version(it1.offset));
```

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 := λ i. i < it1.offset ? i = it1.offset ? t : l.data(i-1);
4:     for all var ∈ V_It/{it1, it2} do
5:         var.offset := var.offset < it1.offset ? var.offset : var.offset+1;
6:         var.version := l.version(var.offset);
7:     it2 := (l̊, it1.offset, l.version(it1.offset));
8:     it1.offset := it1.offset + 1;
9:     it1.version := l.version(it1.offset);
```

Fig. 14. The program I_{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 := λ i. i < it1.offset ? l.data(i) : i = it1.offset ? t : l.data(i-1) ;
4:     l.version := λ i. i < it1.offset ? l.version(i) : l.version(i)+1;
5:     it2 := (l̊, it1.offset, l.version(offset) );
```

Fig. 15. The program I_{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.

	Total	Avg. Time(min)	Min	Max
Failed	33	59.9	37	112
Sucess	103	18.8	< 1	85
Counterexample	3	13.7	6	18

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/

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 components in C. IEEE Transactions on Software Engineering **30** (2004) 388–402
2. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: DAC, ACM (2003) 368–371
3. Ball, T., Rajamani, S.: Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research (2000)
4. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Principles of Programming Languages. (2002) 58–70
5. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. Formal Methods in System Design **25** (2004) 105–127
6. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV. Volume 1254 of LNCS., Springer (1997) 72–83
7. Stepanov, A.A., Lee, M.: The Standard Template Library. Technical Report ANSI X3J16/94-0095, ISO WG21/N0482 (1994)
8. Liskov, B., Zilles, S.: Programming with abstract data types. In: ACM SIGPLAN Symposium on very high level languages, ACM (1974) 50–59
9. ISO/IEC: ISO/IEC 14882:2003 (E). *Programming languages - C++* (2003)
10. Ball, T., Rajamani, S.: Automatically validating temporal safety properties of interfaces. In: SPIN Workshop on Model Checking of Software. (2001) 103–122
11. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Computer Aided Verification. (2000) 154–169
12. Wang, C., Musser, D.: Dynamic verification of C++ generic algorithms. IEEE Transactions on Software Engineering **23** (1997) 314–323
13. Musuvathi, M., Park, D., Chou, A., Engler, D., Dill, D.: CMC: a pragmatic approach to model checking real code. In: Symposium on Operating System Design and Implementation. (2002)
14. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10(2)** (2003) 203–232
15. Hoare, C.A.R., Wirth, N.: An axiomatic definition of the programming language pascal. Acta Informatica **2** (1973)
16. Bornat, R.: Proving pointer programs in hoare logic. In: Mathematics of Program 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 structures. In: Symposium on Principles of Programming Languages. (2001) 14–26
19. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: CAV. Volume 2404 of LNCS. Springer (2002)
20. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Theory and Applications of Satisfiability Testing (SAT). (2003) 502–518