Verifying C++ with STL Containers via Predicate Abstraction

Nicolas Blanc*
Computer Systems Institute, ETH Zurich
blancn@inf.ethz.ch

Alex Groce
Laboratory for Reliable Software, Jet Propulsion Laboratory
agroce@gmail.com

Daniel Kroening†
Computer Systems Institute, ETH Zurich
daniel.kroening@inf.ethz.ch

ABSTRACT

This paper describes a flexible and easily extensible predicate abstraction-based approach to the verification of STL usage, and observes the advantages of verifying programs in terms of high-level data structures rather than low-level pointer manipulations. We formalize the semantics of the STL by means of a Hoare-style axiomatization. The verification requires an operational model conservatively approximating the semantics given by the Standard. Our results show advantages (in terms of errors detected and false positives avoided) over previous attempts to analyze STL usage, due to the power of the abstraction engine and model checker.

Categories and Subject Descriptors: D.2.4[Software Engineering]:Program Verification; D.3.1[Programming Languages]:Formal Definitions

General Terms: Verification, Languages

1. INTRODUCTION

C++ is one of the most widely used programming languages. Software programs including office applications, databases, games, and critical embedded systems are often implemented in C++. Software model checking for C programs is widely recognized as providing real benefits for suitable programs, and is implemented by a number of tools [1, 14, 6]. Previous efforts to model check C++ code are based on explicit execution of the program; we propose to extend the popular *predicate abstraction* framework [12, 2, 7] to the verification of C++ code using abstract data types (ADTs).

We concentrate our efforts on uses of the Standard Template Library (STL) [16], which defines generic containers and iterators. 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 — the STL 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 the STL) allow *programmers* to abstract away from the implementation details of commonly used structures and concentrate on the task at hand [19]. We observe that abstract data types provide the same facility in abstraction for *verification tools*.

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 an operationally equivalent model. SATABS [8, 9] is a predicate abstraction-based model checker for C programs that we extend to handle a large subset of the C++ language, including objects, (operator) overloading, references and templates (without partial specialization). Previous abstraction-based model checkers neither handle C++ programs nor provide an operational semantics supporting implementation-independent verification of code using the STL.

The operational model is an implementation of the Standard Template Library optimized for verification purposes, 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.

Related Work

Wang and Musser present a dynamic approach for verifying template code using `gdb`, which provides correctness proofs only if loop invariants are provided [22]. CMC [20] can, in theory, verify C++ code compiled with templates and STL constructs, but checks implementation-dependent behavior as it actually executes the code. At the other end of the spectrum, SAVCBS 2006 presented iterator specification as a challenge problem, resulting in a number of approaches, focusing mainly on logical specification rather than practical verification methods [17, 23, 3, 18]. Cok shows how to use ESC/Java 2 and JML to verify usage in some cases, but notes the serious limitations of such an approach [10].

Gregor and Schupp [13] describe STLlint, a static analysis tool for checking properties of STL code. Their goals are

*Supported by ETH research grant TH-21/05-1.
†This research is supported by an award from IBM Research.

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.

quite similar to ours: checking the implementation-independent properties of STL usage in source code. STLlint relies on symbolic execution of an executable specification, similar in spirit to our approach, but without a formalization to establish the link between the operational semantics and the STL definition, or the power of model checking to produce *counterexample traces* for errors. The latter is critical both for avoiding spurious errors and for the diagnosis of real errors.

The contribution of this work is twofold: 1) we extend predicate abstraction to C++ programs, and 2) in particular we show that an operational model and the principles of abstract data types can be combined efficiently to verify usage of the C++ STL in an implementation-independent manner.

2. AXIOMATIC SEMANTICS

The C++ standard defines the semantics of the STL informally using pre- and post-conditions — in potentially ambiguous English text that is not machine-readable. We axiomatically formalize the semantics of the standard containers, providing a basis for correctness of an abstract implementation. We define Hoare triples in the "forward"-style for the methods of the container classes. Hoare-style axiomatizations of languages [15] that permit aliasing are problematic [17, 23, 3, 18, 21]; we reduce the aliasing problem between iterators to aliasing between elements of an array, and exclude reasoning about nested data structures. This section provides a short overview of the formalism we developed. Those interested in further details may consult the technical report [4].

We distinguish three types of variables: the set of container variables $\mathcal{C}$, the set of integer variables $\mathcal{N}$, and the set of iterator variables $\mathcal{I}$. By convention, the variable c denotes a container, $\{i, j\} \subset \mathcal{N}$, 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 kinds of container variables: *active* and *inactive* containers. We denote active containers with unprimed variables, e.g., c, and inactive containers by primed variables, e.g., c'. 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:

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

The expression $c.size$ denotes the size of a container c. 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. 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, which corresponds to the syntactic case $\mathcal{C}_{\mathit{IntExpr}}$, denotes the value of the i^{th} element of the container c.

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 $i \in \{0, \ldots, c.size\}$.

We first formalize the concept of the *Iterator*, which is technically a pointer to an element inside a container. Fig. 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 necessary 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-dec*).

As exemplary Hoare's rules, we provide the schemata for the dereference of iterators and the insertion into a list in Fig 2. Note that schema *it-deref-1* can be instantiated if container c is active only. Let l be an active instance of `list<T>`. The *insert* method takes an iterator it_1 and a reference to an object 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. Note that the schema partially defines the semantics of *insert*, as update statements about it_2 and the content of l are omitted. Further rules are presented in the technical report [4].

3. 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.

A container is a tuple $(data[\infty], version[\infty], size)$. A version number is associated with each offset of the data array of a container, and the field *size* encodes the number of elements stored inside the container.

Similarly, an iterator is a tuple (*vcont*, *offset*, *version*). The field $vcont \in \mathcal{A} \dot\cup \{\perp\}$ identifies the container into which an iterator points, or is $\perp$ in the case of an iterator that has not yet been assigned to. The field *offset* records the offset inside the container. Finally, the field *version* is a number used to assert the validity of the iterator.

Our operational model maintains the following invariant: In a state s, an iterator it points into a container c if and

$$\frac{}{c.begin() \overset{0}{\rightarrowtail} c} \quad \text{(it-begin)} \qquad \frac{}{c.end() \overset{c.size}{\rightarrowtail} c} \quad \text{(it-end)} \qquad \frac{it_1 \overset{i}{\rightarrowtail} c \wedge it_2 \overset{i}{\rightarrowtail} c}{it_1 = it_2} \quad \text{(it-eq)}$$

$$\frac{it_1 \overset{i}{\rightarrowtail} c \wedge it_2 \overset{j}{\rightarrowtail} c \wedge i \neq j}{it_1 \neq it_2} \quad \text{(it-neq)} \qquad \frac{it \overset{i}{\rightarrowtail} c \wedge i < c.size}{it + 1 \overset{i+1}{\rightarrowtail} c} \quad \text{(it-inc)} \qquad \frac{it \overset{i}{\rightarrowtail} c \wedge 0 < i}{it - 1 \overset{i-1}{\rightarrowtail} c} \quad \text{(it-dec)}$$

Figure 1: Axiomatization of Iterators

$$\{P \wedge it \overset{i}{\rightarrowtail} c \wedge i < c.size\ \}$$
$$\texttt{t} = *\texttt{it};$$
$$\{P[t/t'] \wedge t = c_i\ \} \qquad c \in \mathcal{A} \quad \text{(it-deref-1)}$$

$$\{\ P \wedge it_1 \overset{i}{\rightarrowtail} l\ \}\ \texttt{it}_2 := \texttt{l.insert}(\texttt{it}_1, \texttt{t});$$
$$\{\ P[l/l'][it_2/it_2'] \wedge i' = i[l/l'] \wedge \quad \text{(lst-ins)}$$
$$\forall it, j < i'\ .\ it \overset{j}{\rightarrowtail} l' \Rightarrow it \overset{j}{\rightarrowtail} l \wedge$$
$$\forall it, j \geq i'\ .\ it \overset{j}{\rightarrowtail} l' \Rightarrow it \overset{j+1}{\rightarrowtail} l\ \}$$

Figure 2: Examples of Hoare's rules

only if the version of the iterator matches the version of the element it points to ($\mathring{c} \in \mathcal{A}$ denotes the variable c itself):

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

Consequently, it is sufficient to increment the version number $c.version[i]$ to invalidate all the iterators pointing to the element $c.data[i]$.

One can show correctness of the operational model with respect to the formal semantics given in Section 2. The following three claims are shown for each of the methods:

1. The invariant is maintained *(ass-ptsto)*,
2. the pre-condition of the operational model is at least as strong as the pre-condition required by the standard,
3. the post-condition of the operational model is at most as strong as the post-condition guaranteed by the standard.

The translation of the operational model for *vector* to C++ is straight forward, and as an illustrating example, we provide here the implementation of the *insert* method:

```
Iterator vector<T>::insert(Iteraror& it, T& t) {
  assert(it.vcont == this);
  assert(it.version == version[it.offset]);
  size++;
  for(int i = size; i > it.offset; i-- )
      data[i] = data[i-1];
  data[it.offset] = t;
  for(int i = it.offset; i <= size; i++ )
     version[i]++;
  return Iterator(this,version[it.offset],
                  it.offset);
}
```

This example omits the consideration of the vector capacity, in the interests of clarity and space. The assertions check whether the iterator `it` is valid or not. Subsequently, the value `t` is inserted into the array. According to the semantics of `vector`, any iterator pointing after the position of insertion needs to be invalidated. This operation is performed by incrementing the version number of all the elements with indexes in range `[it.offset ... size]`. Finally, the method returns a valid iterator pointing to the newly inserted element.

Schema *lst-ins* in Fig. 2 illustrates the difficulties that may arise with the use of universal quantifiers in post-conditions. Due to quantification, every iterator variable pointing after the position of insertion needs to be updated. The operational model overcomes this issue by considering a safe over-approximation, i.e, the checker does not incorrectly report that a program is correct.

A possible over-approximation for a list consists in keeping valid only the iterators whose offsets are not affected by the insertion. The checker may as a result report spurious counterexamples, but the approximation may be sufficient for proving some properties. We hope to implement a refinement procedure for ruling out some spurious counterexamples introduced by the over-approximation.

4. EXPERIMENTAL RESULTS

The operational model uses an unbounded array in order to store the container elements. We extend SATABS in order to support unbounded arrays in the predicates and in the transition relation. This is implemented using a decision procedure for the combined theory of bit-vector arithmetic and array logic. The procedure is similar to the procedure presented in [5].

We use the source code of MiniSat as a benchmark for our technique. MiniSat is "a minimalistic, open-source SAT solver" [11]. The importance of effective SAT solvers to many applications, particularly verification, is well known, and MiniSat is a popular base for research in the field.

We verify the version of MiniSat which uses the vector class provided by the STL. The MiniSat code is hand-crafted for high performance, and makes use of templates, references, and operator overloading.

We obtain a total of 299 non-trivial safety properties for the MiniSat code, out of which 272 are due to the pre-

conditions of our operational version of the vector class. The benchmarks were performed on a Linux machine with a 2.8 GHz Intel Xeon processor. Within 13s (including parsing), our static analysis is able to prove 150 of the properties. The remaining ones are passed to the predicate-abstraction engine. We use a limit of 20 refinement iterations. The times are split up into the time taken by the abstraction, model checking, simulation, and refinement.

		Avg. Time (s)					
	No.	Total	Abs.	Mc.	Sim.	Ref.	It.
CE	5	324.8	18.3	276.7	5.6	23.8	6.4
Success	229	52.8	7.6	38.1	1.5	5.6	2.6
Failed	65	420.6	14.8	331.8	29.1	44.9	20.0

We were able to prove 229 properties (76%) in an average of about one minute each, and obtained counterexamples for 5 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 65 properties, the iteration limit was exceeded. Those cases were mostly due to vector indexing by values taken from non-STL dynamic memory. The model checker and the operational model of STL are available to other researchers for experimentation at `www.verify.ethz.ch/stl/`.

A problematic pattern appearing in MiniSat is the use of a literal taken from a clause vector as an index into another vector, as in this code fragment:

```
Lit q = c[j];
if (!seen[var(q)] && level[var(q)] > 0){
```

where `c` originally derives from a clause database. In order to prove safety for the accesses to `seen` and `level`, SatAbs must derive and manipulate a predicate involving a pair of quantifiers, e.g. $\forall$`i.(i <` `db.size()`$) \Rightarrow (\forall$`j.(j <` `db[i].size()`$) \Rightarrow$ `db[i][j] < seen.size()`) if `c[j]` derives from a vector of clauses `db`. Such a nested quantifier predicate is beyond the state-of-the-art for predicate abstraction tools at this stage. However, we note that the property we really wish to prove is that `var` of a `Lit` is always a valid index into a set of vectors. This is a class invariant of the `Lit` class. As future work in the spirit of our approach to STL, we hope to introduce annotations for such class/type invariants, enabling proofs for programs making judicious use of ADTs.

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 data structures in an implementation-independent manner, leveraging the high-level nature of abstract data types to aid predicate abstraction. The success of this effort demonstrates that ADTs can be as useful in assisting automated reasoning tools in "understanding" code as they are in assisting programmers in organizing code: theorem provers and abstraction engines find ADTs easier to reason about than low-level pointer manipulations, as the implicit relationships in structures are made explicit, in an implementation-independent way. This approach relies on the first reported symbolic model checker for complex C++ code, implemented in the SatAbs model checker.

6. REFERENCES

[1] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, pages 103–122. Springer, 2001.

[2] T. Ball and S.K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research, February 2000.

[3] K Bierhoff. Iterator specification with typestates. In *Specification and verification of component-based systems*, pages 79–82. ACM Press, 2006.

[4] N. Blanc, A. Groce, and D. Kroening. Verifying C++ with STL containers via predicate abstraction. Technical Report 506, Computer Science Department, ETH Zurich, 2006.

[5] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation, (VMCAI)*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer, 2006.

[6] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. on Software Engineering*, 30(6):388–402, June 2004.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*, pages 154–169, 2000.

[8] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design*, 25:105–127, September–November 2004.

[9] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, 2005.

[10] D. R. Cok. Specifying Java iterators with JML and Esc/Java2. In *Specification and verification of component-based systems*, pages 71–74, 2006.

[11] N. Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, 2003.

[12] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[13] D. Gregor and S. Schupp. STLlint: lifting static checking from languages to libraries. *Softw. Pract. Exper.*, 36(3):225–254, 2006.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.

[15] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2, 1973.

[16] ISO/IEC. ISO/IEC 14882:2003 (E). *Programming languages - C++*, 2003.

[17] B. Jacobs, F Piessens, and W. Schulte. VC generation for functional behavior and non-interference of iterators. In *Specification and verification of component-based systems*, pages 67–70. ACM Press, 2006.

[18] N. R. Krishnaswami. Reasoning about iterators with separation logic. In *Specification and verification of component-based systems*, pages 83–86, 2006.

[19] B. Liskov and S. Zilles. Programming with abstract data types. In *ACM SIGPLAN Symposium on very high level languages*, pages 50–59. ACM, 1974.

[20] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: a pragmatic approach to model checking real code. In *Symposium on Operating System Design and Implementation*, 2002.

[21] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic In Computer Science*, pages 55–74. IEEE, 2002.

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

[23] B. W. Weide. SAVCBS 2006 Challenge: Specification of iterators. In *Specification and verification of component-based systems*, pages 75–78, 2006.