20th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems (ECBS)

SMT-Based Bounded Model Checking of C++ Programs

Mikhail Ramalho[1], Mauro Freitas[1], Felipe Sousa[1],
Hendrio Marques[1], Lucas Cordeiro[1], and Bernd Fischer[2,3]
[1] Electronic and Information Research Center, Federal University of Amazonas, Brazil
[2] Electronics and Computer Science, University of Southampton, UK
[3] Department of Computer Science, Stellenbosch University, South Africa
esbmc@ecs.soton.ac.uk

***Abstract*—Bounded model checking of C++ programs presents greater challenges than that of C programs due to the more complex features that the language offers, such as templates, containers, and exception handling. We present ESBMC++, a bounded model checker for C++ programs. It is based on an operational model, an abstract representation of the standard C++ libraries that conservatively approximates their semantics. ESBMC++ uses this to encode the verification conditions using different background theories supported by an SMT solver. Our experimental results show that our approach can handle a wider range of the C++ constructs than existing approaches and substantially reduces the verification time.**

Keywords*—Software engineering, formal methods, verification, model checking.

I. Introduction

Bounded model checking (BMC) based on Boolean satisfiability (SAT) solvers has already been successfully applied to discover subtle errors in real systems [11]. In an attempt to cope with growing system complexity, SAT solvers are increasingly replaced by satisfiability modulo theories (SMT) solvers to prove the generated verification conditions (VCs) [9], [17], [20]. There have also been attempts to apply BMC to the verification of C++ programs [24], [30] but with limited success. The main challenge here is to handle large programs and to support the complex features that the languages offers, such as templates, containers, inheritance, and in particular exception handling, which is an important approach to contain and handle error situations in computer-based systems. At the same time, in order to be attractive for mainstream software development, C++ model checkers have to maintain high speed and accuracy.

Here, we propose to apply SMT-based BMC to C++ programs using an operational model, which is an abstract representation of the standard C++ libraries that conservatively approximates their semantics. We integrate this operational model into our ESBMC model checker [17] that in turn builds on top of CBMC's front-end [16] to support the main C++ features.

We present the implementation of our operational model of the sequential STL containers, its preconditions and simulation features (e.g., how the elements values of the containers are stored), and how these are used in order to verify C++ applications. Additionally, we develop and describe novel approaches to handle exceptions in C++ programs (e.g., exception specification for functions and methods) that previous approaches could not handle [12], [24], [27]. In particular, we implement the inheritance mechanism during the construction of the intermediate representation of the program, which avoids converting the C++ program to a C program and consequently produces smaller models to be verified. Experimental results show that our approach consistently outperforms LLBMC [24], a bounded model checker for C/C++ programs that is also based on SMT solvers.

The remainder of the paper is organized as follows: We first give a brief introduction to the CBMC and ESBMC model checkers and describe the background theories of the SMT solvers that we will refer throughout the paper. In Section III, we describe a simplified representation of the C++ libraries, which conservatively represents the classes, methods, and other features similar to the actual structure. In Section IV, we present our implementation of the inheritance mechanism while Section V is concerned with the implementation of the exception handling approach. In Section VI, we present the results of our experiments using several C++ benchmarks and a real-world C++ application used in the telecommunications domain. In Section VII, we discuss the related work and we conclude and describe future work in Section VIII.

II. Background

ESBMC++ builds on the front-end of CBMC to generate the VCs for a given C++ program, and on the back-end of ESBMC to encode the VCs using different background theories and SMT solvers.

CBMC (C Bounded Model Checker). CBMC implements BMC for ANSI-C/C++ programs using SAT/SMT solvers [16]. It can process the code using the goto-cc tool [29], which compiles the C/C++ code into equivalent GOTO-programs (i.e., control-flow graphs) using a gcc-compliant style. The GOTO-programs can then be processed by the symbolic execution engine. Alternatively, CBMC uses its own, internal parser based on Flex/Bison, to process the C/C++ files and to build an abstract syntax tree (AST). The typechecker of CBMC's front-end annotates this AST with types and generates a symbol table. The intermediate representation (IRep) class of CBMC then converts the annotated AST into an internal, language-independent format used by the remaining phase of the front-end. ESBMC++ modifies this front-end to handle

978-0-7695-4991-0/13 $26.00 © 2013 IEEE
DOI 10.1109/ECBS.2013.15

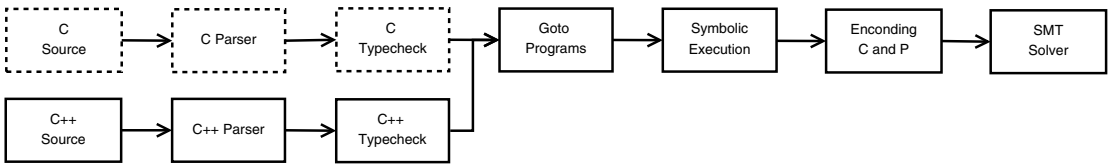

Fig. 1. ESBMC/ESBMC++ Architecture (ESBMC-specific components shown dashed).

the definitions of the standard C++ libraries while the other features (e.g., inheritance, template, and exception handling) are treated internally.

CBMC and the original ESBMC (which builds on CBMC) use two recursive functions $\mathcal{C}$ and $\mathcal{P}$ that compute the *constraints* (i.e., assumptions and variable assignments) and *properties* (i.e., safety conditions and user-defined assertions), respectively. Both tools automatically generate safety conditions that check for example for arithmetic overflow and underflow, array bounds violations, and `NULL`-pointer dereferences, in the spirit of Site's clean termination [28]. Both functions accumulate the control flow predicates to each program point and use these predicates to guard both the constraints and the properties, so that they properly reflect the program's semantics. A VC generator (VCG) then derives the VCs from these.

Satisfiability Modulo Theories. SMT decides the satisfiability of first-order formulae using a combination of different background theories and thus generalizes propositional satisfiability by supporting uninterpreted functions, linear and non-linear arithmetic, bit-vectors, tuples, arrays, and other decidable first-order theories. Given a theory $\mathcal{T}$ and a quantifier-free formula ψ, we say that ψ is $\mathcal{T}$-satisfiable if and only if there exists a structure that satisfies both the formula and the sentences of $\mathcal{T}$, or equivalently, if $\mathcal{T} \cup \{\psi\}$ is satisfiable [13]. Given a set $\Gamma \cup \{\psi\}$ of formulae over $\mathcal{T}$, we say that ψ is a $\mathcal{T}$-consequence of Γ, and write $\Gamma \models_{\mathcal{T}} \psi$, if and only if every model of $\mathcal{T} \cup \Gamma$ is also a model of ψ. Checking $\Gamma \models_{\mathcal{T}} \psi$ can be reduced in the usual way to checking the $\mathcal{T}$-satisfiability of $\Gamma \cup \{\neg\psi\}$.

Arrays and Tuples. The most important theories for ESBMC++ are the array and tuple theories, which are used to model the sequential container data structures and objects, respectively. The array theories of SMT solvers are typically based on the McCarthy axioms [23]. The function *select(a, i)* denotes the value of an array a at index position i and *store(a, i, v)* denotes an array that is exactly the same as array a except that the value at index position i is v. Formally, the functions *select* and *store* can then be characterized by the following two axioms [10], [14], [18]:

$$i = j \Rightarrow \mathit{select}(\mathit{store}(a, i, v), j) = v$$
$$i \neq j \Rightarrow \mathit{select}(\mathit{store}(a, i, v), j) = \mathit{select}(a, j)$$

Array bounds checks need to be encoded separately, as the array theories employ the notion of unbounded arrays size, but arrays in software are typically of bounded size.

Tuples provide store and select operations similar to those in arrays, but work on the tuple elements. Each field of the tuple is represented by an integer constant. Hence, the expression $select(t, f)$ denotes the field f of tuple t while the expression $store(t, f, v)$ denotes a tuple t that at field f has the value v and all other fields remain the same.

Tool Architecture. The tool architecture is shown in Figure 1. The first step is the source code parser; ESBMC++ takes C++ source code as input and creates most of the intermediate representation of the program, which will be the base for the remaining steps of the program verification. The parser is heavily based on the GNU C++ compiler since this allows ESBMC++ to finds most of the syntax errors already reported by GCC.

The next step is the C++ type-check; here, additional checks are performed in the IRep tree, which include assignment checks, type-cast checks, pointer initialization checks, function call checks as well as template creation and instantiation (which will be explained later). In the next step, the IRep tree is converted into goto expressions; this conversion simplifies the representation of the C++ program (e.g, replacement of *switch* and *while* by *if* and *goto* statements), and handles the unrolling of loops and the elimination of recursive functions. In the symbolic execution of the goto programs, the simplified goto program is then converted to SSA expressions and assertions are inserted in the resulting SSA expressions to check for safety properties related to array out-of-bounds, arithmetic under- and over-flow, memory leaks, double frees, and division by zero. Additionally, in this step most of the exception handling is carried out, such as the insertion of GOTO-instructions for the original *throw* statements and exception specification for function calls.

Finally, two set of quantifier-free formulae are created based on the SSA expressions: $\mathcal{C}$ for the constraints and $\mathcal{P}$ for the properties as previously described above. Those two sets of formulae will be used as input for an SMT solver that will produce a counterexample if there is a violation of a given property, or an "unsatisfiable" answer if the property holds.

III. C++ Operational Model

C++ relies on a collection of powerful standard libraries to provide much of the functionality programmers require. In principle, we could use the (available) sources during the verification, but their optimized implementations would complicate the VCs unnecessarily. Instead we developed a simplified representation of the libraries called the C++ Operational Model (COM), which represents the classes, methods, and other features similar to the actual structure [2]. ESBMC++ then relies on the COM, and in particular on the operational model of the standard C++ libraries, to verify properties related to the definitions in the supported data types. In the verification process, the COM libraries thus replace the corresponding actual C++ libraries. The COM consists of four groups of libraries, as shown in Figure 2.

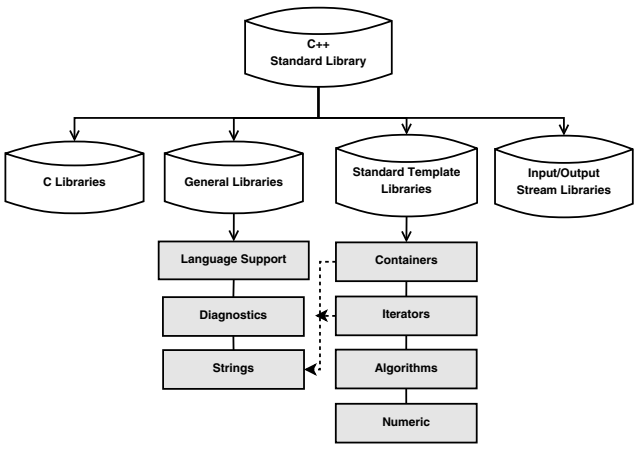

Fig. 2. Overview of the operational model.

Note that the COM also includes the ANSI-C libraries already supported by ESBMC. Since ESBMC++ uses a different front-end (as shown in Figure 1), we have to build a representation of the ANSI-C libraries into the COM; otherwise, ESBMC++ would not recognize the library methods and fail to parse the C++ programs. However, the biggest part of the COM models the Standard Template Libraries (STL). This part is split into four categories: *algorithms*, *numeric*, *containers*, and *iterators*. In this paper, we focus on the operational model of the sequential containers and iterators in the STL, and how they are used to verify real-world C++ programs.

Apart from the STL handling, the verification of C++ programs with templates is essentially split into two steps: template creation and template instantiation. To create a template, ESBMC++ finds the respective template declaration and creates an internal representation of the class or function by flagging the types as generic; no other representation is created here since at this step ESBMC++ does not know which types will be instantiated. To instantiate a template, ESBMC++ finds a template usage with a specified type and creates a new internal representation of the class or function with the instantiated type; this new representation is not a template anymore. At this point, ESBMC++ keeps track of the generic template definition and the respective instantiated class or function. Note that when a new template is instantiated, ESBMC++ first checks whether it was already instantiated to avoid creating a duplicate representation of a previously instantiated template.

A. Core Container Language

To formalize the verification of the STL containers, we define a core container language, and extend the translation functions $\mathcal{C}$ and $\mathcal{P}$ of constraints and properties to this. We then use this core language to implement the operational model of the containers.

The container language comprises several syntactic domains, starting with the base elements T, iterators It, pointers P, and integer indices Int, and of course the (proper) container expressions C. Figure 3 summarizes the core container syntax. Here t, i, p, and c are variables of type T, It, P, and C, respectively. n is a variable or constant of type Int. We abuse

$$
\begin{aligned}
T &::= t \mid *It \mid *P \\
It &::= i \mid C.begin \mid C.end \\
&\quad \mid C.insert(It, T, N) \mid C.insert(It, It, It) \\
&\quad \mid C.erase(It) \mid C.erase(It, It) \mid C.search(It) \\
P &::= p \mid P(+ \mid -)P \mid C.array \\
Int &::= n \mid Int(+ \mid * \mid \ldots)Int \mid It.pos \mid C.size \mid C.capacity \\
C &::= c \mid It.source
\end{aligned}
$$

Fig. 3. Core container syntax.

the notation $*It$ to denote the value stored in the underlying container at the position pointed to by the iterator It; this is an abbreviation for $(It.source.array)[It.pos]$. $*P$ is the value stored in the P position of the memory.

$C.begin$ and $C.end$ are methods that return iterators which point to the beginning and the ending of a container, respectively. Most container operations also return an iterator pointing to the new focus element after the operation rather than simply returning an updated container. For example, for vectors $C.erase$ returns an iterator pointing to the right neighbor of the erased element. Note that the only way in the core language to access the resulting container is thus via the $source$ field of the returned iterator.

Finally, $C.array$ is a memory address that stores the beginning of the container array, $It.pos$ is the index (within this array) of the element that an iterator points to, and $C.size$ and $C.capacity$ return the actual and maximum size, respectively, of the containter C.

B. Operational Container Model

As the container structures differ slightly from each other, some of their methods will vary too, changing the internal models as well (e.g., a *list* container does not have a reference operator and its elements are only reached by iterators).

To simulate the containers appropriately, our model makes use of three variables: a variable of type P called *array* that points to the first element of the array, a natural number *size* that stores the quantity of elements in the container, and a natural value *capacity* that stores the total capacity of a container (which is valid only for vectors). Note that, as the elements are added to a vector container and the size grows, the capacity is doubled every time the size reaches the existing capacity value. Similarly, iterators are modeled using two variables: a variable of type *Int* called *pos*, which contains the index value pointed by the iterator in the container and a variable of type P called *source*, which points to the underlying container. Figure 4 gives an overview of our operational model for the STL sequantial containers.

The core container language only supports the methods listed in Fig. 3. Other methods such as *push_back()*, *pop_back()*, *front()*, *back()*, *push_front()*, and *pop_front()* are only a simplified variation of those main methods, which are optimized for some containers (e.g., popping the last element of a *stack*). As part of the single static assignment (SSA) transformation, side-effects on the iterators and containers are made explicit, so that operations return new iterators and containers as result. As an example, consider a container c

with the method call *c.insert* that returns an iterator result and makes use of an iterator i that points to the desired (insertion) position; a template value t with the element to be inserted and an integer n that denotes the number of times the element is to be inserted. The statement $c.insert(i, t, n)$; (which discards the returned iterator) thus becomes $(c', i') = c.insert(i, t, n)$; (where the side effects are explicit).

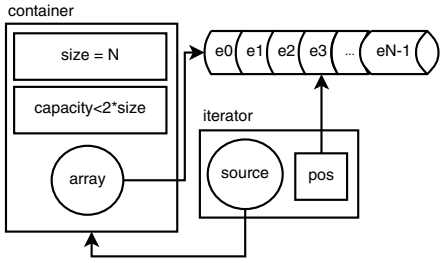

Fig. 4. Operational model of the STL sequential containers.

The translation function $\mathcal{C}$ describes the constraints relating the "before" and "after" versions of the respective model variables. In particular, we get:

$$
\begin{aligned}
&\mathcal{C}((c', i') = c.insert(i, t, n)) := \\
&\quad \wedge\, c'.size = c.size + n \\
&\quad \wedge\, c'.array = store(\ldots(store(\\
&\qquad\qquad store(\ldots(store(c.array, i.pos, t), \\
&\qquad\qquad\quad \ldots, \\
&\qquad\qquad\quad i.pos + n - 1, t), \\
&\qquad\qquad i.pos + n, select(c.array, i.pos)), \\
&\qquad\qquad \ldots, \\
&\qquad\qquad c.size + n - 1, select(c.array, c.size - 1)) \\
&\quad \wedge\, i'.source = c' \\
&\quad \wedge\, i'.pos = i.pos + n
\end{aligned}
$$

The main effect of the *insert* method is thus captured by the second equality that describes the contents of the container array $c'.array$ after the insertion in terms of update operations to the container array $c.array$ before the insertion.

There is another version of the insert method. Here it is possible to insert a sequence of elements in the desired insertion position, using both iterator (or even pointer) bounds to select the sequence from another container. Let i_0 be an iterator that marks the first element to be inserted, i_k be another iterator that points to the first element after the end of the sequence to be inserted in the required position and let k be the length of the array $[i_0\, i_k)$. Thus, we have:

$$
\begin{aligned}
&\mathcal{C}((c', i') = c.insert(i, i_0, i_k)) := \\
&\quad \wedge\, k = i_k.pos - i_0.pos + 1 \\
&\quad \wedge\, a = i_0.source.array \\
&\quad \wedge\, c'.size = c.size + k \\
&\quad \wedge\, c'.array = store(\ldots(store(\\
&\qquad\qquad store(\ldots(store(c.array, \\
&\qquad\qquad\quad i.pos, select(a, i_0.pos)), \\
&\qquad\qquad\quad \ldots, \\
&\qquad\qquad\quad i.pos + k - 1, select(a, i_k.pos - 1)), \\
&\qquad\qquad i.pos + k, select(c.array, i.pos)), \\
&\qquad\qquad \ldots, \\
&\qquad\qquad c.size + k - 1, select(c.array, c.size - 1)) \\
&\quad \wedge\, i'.source = c' \\
&\quad \wedge\, i'.pos = i.pos + k
\end{aligned}
$$

Note that this also implicitly induces the two properties that $[i_0\, i_k)$ is non-empty (i.e., that $k > 0$ holds) and that i_0 and i_k are iterators over the same underlying container (i.e., that $i_0.source.array = i_k.source.array$ holds), although this container can be different from the one we are inserting into.

The erase method works similarly to the insert method. It also uses iterator positions, integer values, and pointers, but it does not use values since the exclusion is made by a given position, regardless the value. It also returns an iterator position, pointing to the position next to the previously erased part of the container. The following model shows the *erase* method that deletes a single element:

$$
\begin{aligned}
&\mathcal{C}((c', i') = c.erase(i)) := \\
&\quad \wedge\, c'.size = c.size - 1 \\
&\quad \wedge\, c'.array = store(\ldots(store(c.array, \\
&\qquad\qquad i.pos, select(c.array, i.pos + 1)), \\
&\qquad\qquad \ldots, \\
&\qquad\qquad c.size - 2, select(c.array, c.size - 1)) \\
&\quad \wedge\, i'.source = c' \\
&\quad \wedge\, i'.pos = i.pos
\end{aligned}
$$

Note that this implicitly induces the property that i is an iterator over c (i.e., that $i.source = c$ holds).

It is also possible to delete a number of elements from the container by marking the bounds with iterators. It works similarly to the equivalent *insert* method; the details are omitted here. Searches are made in a container by using reference operators and a pointing type (pointer or iterator), and return the reference value (the element stored itself).

IV. Inheritance and Polymorphism

C++ features like inheritance and polymorphism makes static analysis difficult to implement. In contrast to Java, which only allows single inheritance, where derived classes have only one base class, C++ also allows multiple inheritance, where a class may inherit from one or more unrelated base classes. This particular feature makes C++ programs harder to model check than programs in other object-oriented programming languages (e.g., Java) since it disallows the direct transfer of techniques developed for other, simpler programming languages [25], [26].

To deal with inheritance in ESBMC++, we simply replicate the methods and attributes of the base classes to the inherited class to have direct access to them. If a class inherits from a base class that does not contain virtual methods, then we call this *replicated inheritance*. If there is a path from class X to class Y whose first edge is virtual, then we call this *shared inheritance*.

A formal description to represent the relationship between classes can be described by the class hierarchy graph (CHG). This graph is represented by a triple $\langle C, \prec_s, \prec_r \rangle$, where C is the set of classes, $\prec_s \subseteq C \times C$ refers to *shared inheritance* edges, and $\prec_r \subseteq C \times C$ are *replicated inheritance* edges. We also define $\prec_{sr} = \prec_s \cup \prec_r$ and $\leq_{sr} = (\prec_{sr})^*$. $(C, \leq_{sr})$ is then a partially ordered set [22] and $\leq_{sr}$ is anti-symmetric (i.e., if one element A of the set precedes B, the opposite relation cannot exist). As an example, Figure 5 shows an UML diagram that represents the *Shape* class

hierarchy that contains multiple inheritance. The replicated inheritance in the Rectangle class relation can be formalized by $\langle C, \emptyset, \{(Rectangle, Shape), (Rectangle, Display)\}\rangle$.

Our tool creates an intermediate model for single and multiple inheritance, handling replicated and shared inheritance where all classes are converted into structures and all methods and attributes of its parent classes are joined. On the one hand, this approach has the advantage of having direct access to the attributes and methods of the derived class and thus allows an easier validation, as the tool does not search for attributes or methods from base classes on each access. On the other hand, we replicate information to any new class, thus wasting memory resources.

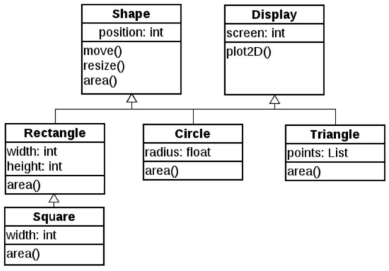

Fig. 5. *Shape* class hierarchy UML diagram

Another important feature from object-oriented programming that we support is the concept of polymorphism, which allows the creation of reusable code by changing only specific features from the base class. Polymorphism allows variable instances to be bound to references of different types according to the structure of the inheritance hierarchy [8]. We thus consider that two or more derived classes from the same base class can invoke methods with the same signature but with distinct behaviors, specialized for each derived class, using for this one reference to each object of this base class type. The decision of which method must be used cannot be made at compile-time. One solution is the usage of virtual tables (described below) that contains the object's method address. In this case, the method call will fetch the correct method address from the object's dispatch table at verification-time.

The intermediate representation of C++ programs in ESBMC++ provides a model to handle polymorphism so that we can simplify the class hierarchy, thus easing the access to methods with the same name without ambiguity between base and derived classes. As an example, our tool can easily handle the polymorphic area method using this representation, as shown in Formulas (1) and (2), using the background theories.

$$C := \left[\begin{array}{l} j_1 = store(j_0, vtable, Rectangle) \\ \wedge\, j_2 = store(j_1, width, 10) \\ \wedge\, j_3 = store(j_2, height, 10) \\ \wedge\, j_4 = store(j_3, vtable, Square) \\ \wedge\, j_5 = store(j_4, width, 10) \\ \wedge\, return\_value_1 = \\ \quad select(j_5, width) \times select(j_5, width) \end{array}\right], \quad (1)$$

$$P := [\ return\_value_1 = 100\ ] \quad (2)$$

The classes *Rectangle* (which is the base class) and *Square* (which is the derived class) each have a virtual method called $area()$, which have the same signature. If we assume that the example calls this method on a base class pointer, then the actually executed function cannot be determined at compile-time. To overcome this problem, we thus create a *vtable* to contain the address of the object's bound methods so that the call to this method is fetched with the address from the *vtable* at verification-time.

In addition, we also support indirect inheritance, where a class inherits features from a derived class with one or more classes not directly connected. In Figure 5, we have $Square \leq_{sr} Rectangle$ and $Rectangle \leq_{sr} Shape$. Thus, the *Square* class can access features from the *Shape* class, but they are not directly connected. We tackle this problem by looking for the features using a depth search from the derived to base classes and adding them to our intermediate representation if necessary.

In OO programming, the use of *shared inheritance* is very common. In contrast to other approaches (e.g., [12]), ESBMC++ is able to verify this kind of inheritance. If a class has pure virtual methods only, then this class does not contain any implementation for these methods and they will thus be implemented in the derived classes. Otherwise, if a class has only virtual methods, it must contain an implementation for them or the verification will fail with a "conversion error". ESBMC++ also handles virtual destructors successfully and supports the default constructor creation. Currently, ESBMC++ supports dynamic cast between primitive types, same classes and from a derived class to a base. ESBMC++ also handles with cast to a reference type, verifying the correct use of *bad_cast* thrown by dynamic cast.

V. Exception Handling

One of the main features that C++ provides is exception handling. The exception handling is split into three elements: a *try* block, where an exception may occur; a *catch* block (also called *handler*), where an exception can be handled; and a *throw* expression to connect both blocks. Figure 6 shows a C++ code example with exception handling.

```
1  int main() {
2    // try block
3    try {
4      throw 20; // throw expression
5    }
6    // catch block
7    catch (int i) {
8    /*error handling for int exceptions*/
9    }
10   catch (float f) {
11   /*error handling for float exceptions*/
12   }
13   return 0;
14 }
```

Fig. 6. Try-catch example: Throwing an integer exception.

In ESBMC++, the exception handling happens in two steps, during the type-checking and the symbolic execution phases. In the type-checking phase, an AST is built based on the code inside the try block, but with a few modifications: before the try

block, a CATCH instruction with an empty map (which will be filled later during type-checking) is inserted, followed by the respective code inside the try block. Here, another CATCH instruction (to represent the end of the try block and the beginning of the catch block) is inserted together with a GOTO instruction, which points to the code after the catch block. This GOTO instruction will only be modified if an exception is thrown; otherwise it will remain the same. After type-checking the try block, ESBMC++ type-checks the handler, which might contain one or more catch blocks. Again, the AST will be created based on the code inside the catch blocks, but with one modification: a GOTO instruction is inserted at the end of each catch block, which points to the code after the catch blocks. Each catch block will thus be assigned a label so that ESBMC++ can decide which catch should be called during the symbolic execution phase if an exception is thrown. At the end of the catch block, the map of the first CATCH instruction is inserted before the try block code is filled with the label created for each catch mapped on the type of the exception. Figure 7 shows the internal flow of ESBMC++ for the exception handling of the code shown in Figure 6.

```
1  ...
2  CATCH signed_int->1, float->2
3  THROW 20
4  $TARGET = 3;
5  if(THROW_TYPE == signed_int)
6    $TARGET = 1
7  else if(THROW_TYPE == float)
8    $TARGET = 2
9  CATCH
10 GOTO $TARGET
11 1: int i;
12    /*error handling for int exceptions*/
13    GOTO 3
14 2: float f;
15    /*error handling for float exceptions*/
16 3: return 0;
```

Fig. 7. Try-catch conversion to goto functions.

During the symbolic execution phase, when the first CATCH instruction is found, the catch map is stacked for later usage. The idea behind the use of a stack is that we may have try-catch blocks inside other try-catch blocks and ESBMC++ should always handle the most internal first. Following the symbolic execution for the code that is inside the try block, ESBMC++ will continue to execute the code until it finds a THROW expression. When it happens, ESBMC++ looks at the map for a valid catch for the exception thrown; if it finds a valid catch, then the label will now be saved, but it will only be handled later; if it is unable to find an exception, then an error will be thrown. ESBMC++ will also ignore any other THROW or GOTO instruction after the first THROW is found, but it will continue to verify all the try block code. When the second CATCH is found, which means that the try block ended, the catch map is unstacked for memory efficiency and the GOTO instruction is thus updated (if needed).

A. Throwing and Catching an Exception

In addition to explicitly throwing an exception, several situations in C++ code can also implicitly cause an exception to be thrown: (a) the operator *new* can throw a *bad_alloc* exception; (b) the operator *dynamic_cast* can throw a *bad_catch* exception; and (c) the function *typeid* can throw a *bad_typeid* exception. In the C++ standard [1], several rules are defined of how any exception thrown is connected to a catch block. In summary, every time when an exception is thrown and one of the following rules is true, the code jumps from the throw expression to the catch block as follows:

1) The handler that catches the exception is the first catch with a matching type; we thus maintain a list with the order of catches and get the catch with the lowest value.
2) A handler will catch an exception thrown if the type thrown and the type of the handler are the same (ignoring const-volatile qualifiers): Here, we simply look for the type of the exception in the catch map and then update the GOTO instruction accordingly if we find a match, or we simply return an error otherwise.
3) Throwing exceptions of types “arrays of type T” and “functions returning type T” will be caught by handlers with “pointer to type T” and “pointer to function returning type T” types: Here, the conversion is made on the type-checking and the throw expression throws two exceptions: “array of type T” and “pointer of type T”, and “function returning type T” and “pointer to function returning type T”, respectively. The handler that catches the exception thrown is determined by the first rule in cases of multiple matches.
4) The handler will catch an exception of type T if the handler type is an unambiguous public base class of T: The conversion is similar to the previous rule, but here several exceptions may be thrown: the type of the object and the type of its bases. Again, the handler will be determined by the first rule in cases of multiple matches.
5) The handler will catch an exception of type pointer T if T’s type can be converted to the type of the handler, either by qualification conversion or standard pointer conversion: Similar to the previous rules, on the type-checking phase the possible conversions based on the catches types will be thrown with the original pointer type, with the handler being determined by the first rule in cases of multiple matches.
6) If the exception thrown is a pointer, then a handler with type *void** or *nullptr_t* can catch it: during the symbolic execution, if no match is found in the map and the exception thrown is a pointer, we simply look for a *void** or *nullptr_t* catch and then update the GOTO instruction. If the exception had a match, then this rules is ignored.
7) A handler of type ellipsis (...) will catch any thrown exception, and shall be the last handler on the catch block: This is similar to the previous rule, but here it works for every type; if no match is found, ESBMC++ looks for a handler of type ellipsis and updates the GOTO instruction accordingly if one exists.
8) If the throw has no arguments, then it should rethrow the last thrown exception: we always keep a reference of the last thrown exception and then update a rethrow

if this reference is not NULL.

B. Exception Specification

The exception specifications define which exceptions a function or method (including constructors and destructors) can throw. For each method or function declaration the exception specification lists the exceptions that can "escape" the respective function or method, i.e., are not guaranteed to be handled within. Note that an exception can still be handled inside a try-catch block inside the function or method even if it is not listed in the exception specification.

The exception specification is handled by ESBMC++ by inserting a THROW_DECL instruction after the declaration of each function or method. In the symbolic execution phase, the exception specification is stacked and removed in the END_FUNCTION instruction at the end of every function or method. The idea for stacking the exception specification is the same as for catch maps, ESBMC++ may find function calls to other function and they may also have their own exception specifications. Finally, when an exception is thrown, ESBMC++ checks whether there is an exception specification currently in force and, if so, whether the exception thrown is allowed to be thrown outside the function. If it is allowed, the exception handling follows and tries to look for an match on the catch map; otherwise it will return an error.

VI. Experimental Evaluation

This section is split into three parts. The setup is described in Section VI-A while Section VI-B describes a comparison between ESBMC++ [3] and LLBMC (Low-Level Bounded Model Checker) [4] using a set of standard C++ benchmarks. Some details about LLBMC are also given in Section VI-B. In our experiments, we also tried to use the CBMC model checker [16], but since it has failed in most of our benchmarks (as reported previously by Merz et al. [24]), we do not report any results. In Section VI-C, we describe the results of verifying a commercial application from the telecommunications domain using ESBMC++.

A. Experimental Setup

The benchmarks that are used in our comparison consist of 1113 C++ programs. Around 290 programs are extracted from Deitel's textbook [19], 16 programs are taken from the NEC benchmark suite [6], 16 programs are taken from the LLBMC benchmark suite [27], and the others were developed by us to test all the features that the C++ language provides. The benchmarks are split into eleven suites, as follows: *algorithm* contains test cases for methods that involve the algorithm library; *cpp* contains general test cases of the C++ language that involve the general libraries, multi-threading, and templates. Additionally, it also contains the LLBMC benchmarks and most of the Deitel benchmarks. The categories *deque*, *list*, *queue*, *stack*, *stream*, *string*, and *vector* contain test cases for the respective STL container structures. Finally, *inheritance* contains test cases related to inheritance and polymorphism while *try_catch* contains test cases related to exception handling; the NEC test cases are located in this suite.

All the experiments were conducted on an otherwise idle Intel Core i7-2600, 3.40 GHz with 24 GB of RAM running Ubuntu 64-bits. For all test suites the individual time limit and memory limit for each test has been set to 900 seconds and 24 GB (22 GB of RAM and 2 GB of virtual memory), respectively. The times given were measured using the *time* command.

B. Comparison to LLBMC

This subsection describes the evaluation of ESBMC++ against LLBMC, another C++ BMC tool developed by Merz et al. [24]. Table I summarizes the results. Here, *N* is the number of C++ programs, *L* is the total lines of code of each suite, *Time* is the total verification time of each suite, *P* is the number of correct positive results (i.e., the tool reports SAFE correctly), *N* is the number of correct negative results (i.e., the tool reports UNSAFE correctly), *FP* is the number of false positive results (i.e., the tool reports SAFE incorrectly), *FN* is the number of false negative results (i.e., the tool reports UNSAFE incorrectly), *Fail* is the number of internal errors during the verification of each suite, *TO* represents the number of time-outs (i.e., the tool was aborted after 900 seconds), and *MO* represents the number of memory-outs.

We invoked both tools using two scripts: one for ESBMC++, that reads the parameters from a file and calls the tool,[1] and another for LLBMC that first compiles the code to bytecode using CLANG [5],[2] reads the parameters from a file and calls the tool.[3] The bound set for both tools (value of *B*) depends of each test case. LLBMC currently does not support exception handling and all the bytecodes were generated without exception support (flag -fno-exceptions) while verifying with LLBMC. Enabling exceptions resulted in LLBMC aborting in most of the cases.

As we can see in Table I, LLBMC times out in 24 programs in the *algorithm* suite and runs out of memory in two programs. If we carefully analyze those test cases, most of them use iterators, which might be causing the slow down in the verification process; this is a situation that also happens in other suites. In the *deque*, *vector*, and *list* suites, the slowdowns still happen but with small values. The suite that had the most unsuccessful verification results was the *list* suite, and most of the errors were related to the container size (e.g., assertions if the container is empty or if it has a particular size). In ESBMC++, most of the errors on those suites are due to a missing operational model of the libraries, which are currently under development.

In the *queue* suite, LLBMC fails in a program that uses the size of a list as constructor parameter while in the *stack* suite all programs are correctly verified. In ESBMC++, all the programs in both suites are successfully verified.

In the *stream* suite, most of the errors are related to assertions on the size of the stream (using the method $gcount()$) and to internal flags (such as *ios::hex* and *iostream::hex*). In ESBMC++, most of the error are related to a bad operational model of the internal flags. In the *string* suite, the errors are related to assertions in the string itself, usually if the string

[1] esbmc --unwind *B* --no-unwinding-assertions -I /libraries/ –timeout 15m

[2] /usr/bin/clang++ -c -g -emit-llvm *.cpp -fno-exceptions /usr/bin/llvm-link *.o -o main.bc

[3] llbmc --ignore-missing-function-bodies --no-max-loop-iterations-checks --max-loop-iterations=*B*

	Testsuite	N	L	ESBMC++								LLBMC							
				Time	P	N	FP	FN	Fail	TO	MO	Time	P	N	FP	FN	Fail	TO	MO
1	algorithm	130	3376	996	63	38	16	13	0	0	0	22964	53	45	1	5	0	24	2
2	deque	43	1239	238	19	20	0	4	0	0	0	8585	16	17	0	0	1	9	0
3	vector	146	6853	2714	95	37	3	11	0	0	0	7234	91	38	1	3	4	6	3
4	list	70	2292	3928	25	25	3	17	0	0	0	2562	5	26	5	30	0	0	4
5	queue	14	328	177	7	7	0	0	0	0	0	45	6	7	0	1	0	0	0
6	stack	12	286	82	6	6	0	0	0	0	0	45	6	6	0	0	0	0	0
7	inheritance	51	3460	311	28	17	1	2	3	0	0	122	32	12	1	3	3	0	0
8	try_catch	67	4743	45	17	41	7	2	0	0	0	4	0	1	0	0	66	0	0
9	stream	66	1831	1892	51	13	0	2	0	0	0	11	17	13	0	35	1	0	0
10	string	233	4921	46186	100	112	5	16	0	0	0	37	6	121	4	102	0	0	0
11	cpp	343	26624	1817	269	38	7	25	4	0	0	3260	235	24	10	56	15	2	1
		1175	55953	58386	680	354	42	92	7	0	0	44869	467	310	22	235	90	41	10

TABLE I. RESULTS OF THE COMPARISON BETWEEN ESBMC++ V1.20 AND LLBMC V2012.2A.

is equal to another string. In the *inheritance* suite, LLBMC reports incorrect errors about memory writing and instantiation of virtual methods (that do not contain implementation). It also does not support some expressions in the SMT back-end (e.g., "Op % (nondef) found"). ESBMC++ fails to verify test cases related to the use of the *dynamic_cast* (as described in Section IV).

In the *try_catch* suite, LLBMC failed in most of the tests due to the fact that the tool is missing support to exception handling. ESBMC++ was able to verify most of the cases. The errors that occur are related to a missing implementation of exception specifications when using classes constructors. And lastly, in the *cpp* suite, which has test cases involving all the other suites (but are not redundant), most of the errors presented were already seen during the verification of other suites.

ESBMC++ verified all suites in 58386 seconds (approximately 16 hours) and gave the right results for 1034 out of 1175 programs (88%) while LLBMC verified all suites in 44869 seconds (approximately 12 hours) and gave the right results for 777 out of 1175 programs (66%). We can see that LLBMC is slower than ESBMC++ on most of the containers and *algorithm* suites, while it is faster on *stream* and *string* suites but looses on successfully verified test cases. In the *inheritance* suite, the results of both tools are essentially the same. In the *try_catch* suite, ESBMC++ is able to verify almost all programs, something that LLBMC cannot due to its lack of support of exception handling. In the *cpp* suite, ESBMC++ is able to successfully verify more programs than LLBMC. Note that ESBMC++ does runs out of memory or time in any suite.

C. Verifying the Sniffer Code

This section describes the results of the verification process using the ESBMC++ and LLBMC model checkers against the sniffer code provided by Nokia Institute of Technology (INdT). The sniffer code is responsible for capturing and logging traffic passing over a network that supports the Message Transfer Part Level 3 User Adaptation Layer (M3UA); it enables the transport of Signaling System 7 (SS7) protocol's user parts and it uses the services provided by the Stream Control Transmission Protocol (SCTP). The sniffer code contains approximately 20 classes, 85 methods, and 2800 lines of C++ code.

The following properties were verified using the customer version of the sniffer code: array bounds violations, division by zero, and arithmetic under- and over-flow. Due to confidentiality issues, we were able to model check 50 out of 85 methods (since we did not have access to some external classes that the sniffer code requires). In the remaining code base, ESBMC++ was able to identify five bugs that are mostly related to arithmetic under- and over-flow while LLBMC was able to identify only three of them. Note that all bugs were reported to the developers and confirmed by them.

As an example of the bugs that were found, Figure 8 shows a code fragment of the method *getPayloadSize* from the class *PacketM3UA*. Here, an arithmetic over-flow might occur on the typecast operation since the method *ntohs* returns an unsigned integer, but the method *getPayloadSize* is expected to return an integer data-type. One possible way to fix this bug is to change the return type of the method *getPayloadSize* to unsigned integer to avoid the typecast over-flow.

```
1 int PacketM3UA::getPayloadSize() {
2   return ntohs(m3uaParamHeader->paramSize)
3             - (M3UA_PROTOCOL_DATA_HEADER_SIZE
4             + M3UA_PARAMETER_HEADER_SIZE);
5 }
```

Fig. 8. Arithmetic over-flow on the typecast operation of the *getPayloadSize*.

VII. RELATED WORK

The application of SMT-based BMC to software is gaining popularity in the software engineering community mainly due to the advent of sophisticated SMT solvers built over efficient SAT solvers [10], [14], [18]. Previous work related to SMT-based BMC for software addresses the problem of verifying C programs that use bit operations, floating-point arithmetic, and pointers [16], [9], [20], [17]. However, there is only little work that addresses the problem of model checking C++ programs that make use of templates, containers, and exception handling.

Prabhu et al. [27] present an interprocedural exception analysis and transformation framework for C++ that records the control-flow created by the exceptions and creates an exception-free program. The exception-free program creation

starts by generating a modular interprocedural exception control-flow graph (IECFG). The IECFG is refined using an algorithm based on a compact representation for a set of types called the Signed-TypeSet domain and the result is used to generate the exception-free program. Finally, the exception-free program is verified using F-SOFT [21]. The verification is focused on two properties: "no throw", the percentage of the code that does not raise an exception and "no leak", the number of memory leaks on try-catch blocks [27].

Jing Yang et al. present a translation tool called Class Hierarchy Representation Object Model Extension (CHROME) that is targeted towards making static program analyzers for C++ easier to write and provide more precise results [30]. CHROME makes a source-to-source transformation from a C++ program with inheritance into a semantically equivalent program without inheritance by treating the inheritance with separate memory regions that are linked to each other via additional base class and derived class pointer fields. This transformation comprises a clarifier, which makes implicit C++ features explicit. This approach was also implemented with F-SOFT [21]. CHROME has a different memory behavior from the original program and therefore does not allow the use of low-level primitives (e.g, *memset*). The CHROME-lowered C program is three to five times bigger than the size of the original C++ program.

Blanc et al. describe the verification of C++ programs that use the STL containers via predicate abstraction [12]. They make use of abstract data types for the STL usage verification rather than the actual STL implementation and behavior. Blanc et al. show that it suffices to verify correctness using an operational model by proving that the pre-conditions on operations in the model imply the pre-conditions guaranteed by the language definition for those operations; similarly, the post-conditions given by the standard imply the strongest post-conditions for the operational model. This approach is efficient in finding trivial errors in C++ programs, but it lacks on a deeper search for bugs and misleading operations (i.e, when it involves internal modeling of the methods).

Merz et al. describe the LLBMC tool, which also applies BMC to the verification of C++ programs [24]. However, they use the LLVM compiler to convert C++ programs into LLVM's intermediate representation, which thus looses high-level information about the structure of the C++ programs (i.e., the relationship between the classes). Similarly to ESBMC++, Merz et al. also apply SMT solvers to check the verification conditions that are generated from the C++ programs. In contrast to our approach, however, they do not handle exceptions, which thus make it difficult to verify realistic C++ programs (e.g., programs that depend on the STL library).

Java PathFinder is an explicit-state model checker for Java programs, but Pasareanu and Visser [25] also developed a symbolic execution framework for it. However, due to the considerable differences between Java and C++ it is difficult to compare this to ESBMC++.

VIII. Conclusions

In this work, we have investigated SMT-based verification of C++ programs by focusing on the major features that the language offers. We have described an implementation of an operational model of the sequential STL containers as well as novel approaches to handle inheritance, polymorphism, and exception handling (in particular, exception specification, which is a feature that is not supported by other BMC tools). Our experiments contain C++ programs with most of the features that the C++ language has to offer. Additionally, we have verified a commercial application of medium-size used in the telecommunications domain. The results show that ESBMC++ outperforms LLBMC for the verification of C++ programs. In particular, ESBMC++ is able to verify most of the C++ programs; we are able to verify programs with exceptions enabled (a missing feature of LLBMC that decreases the verification accuracy of C++ programs). In addition, ESBMC++ was able to find undiscovered bugs in the sniffer code that were later confirmed by developers. For future work, we intend to extend the operational model of STL containers to support not only sequential containers, but also mapped ones (e.g., map and multimap).

Acknowledgment

The development of ESBMC++ is funded by the Royal Society and by Nokia Institute of Technology (INdT).

References

[1] Working draft, Standard for Programming Language C++, http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2012/n3376.pdf, 2012.

[2] Reference of the C++ Language Library, http://www.cplusplus.com/reference/, 2012.

[3] Efficient SMT-Based Context-Bounded Model Checker, http://esbmc.org/, 2012.

[4] The Low-Level Bounded Model Checker, http://llbmc.org/, 2012.

[5] LLVM Tools, http://llvm.org/releases/, 2012.

[6] NEC, http://www.nec-labs.com/research/system/, 2012.

[7] SMT-LIB, http://combination.cs.uiowa.edu/smtlib, 2009.

[8] R. T. Alexander, J. Offutt, and J. M. Bieman. Fault Detection Capabilities of Coupling-based OO Testing In *ISSRE* pp. 207–2002, 2002.

[9] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *STTT*, vol. 11 (1), pp. 69–83, 2009.

[10] C. Barrett and C. Tinelli, CVC3. In *CAV*, LNCS 4590, pp. 298–302, 2007.

[11] A. Biere. Bounded model checking. In *Handbook of Satisfiability*, pp. 457–481. 2009.

[12] N. Blanc, A. Groce, and D. Kroening, Verifying C++ with STL containers via predicate abstraction. In *ASE*, pp. 521–524. 2007.

[13] A. R. Bradley and Z. Manna. The Calculus of Computation: Decision Procedures with Applications to Verification. Springer, 2007.

[14] R. Brummayer and A. Biere, Boolector: An efficient SMT solver for bit-vectors and arrays. In *TACAS*, LNCS 5505, pp. 174–177, 2009.

[15] A. Cimatti et al. Verifying SystemC: a software model checking approach. In *FMCAD*, pp. 121–128, 2010.

[16] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, *LNCS* 2988, pp. 168–176, 2004.

[17] L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In *IEEE Trans. Software Eng.*, v. 38, n. 4, pp. 957–974, 2012.

[18] L. M. de Moura and N. Bjørner, Z3: An efficient SMT solver. In *TACAS*, LNCS 4963, pp. 337–340, 2008.

[19] P. Deitel and H. Deitel. C++ How to Program. Prentice Hall, 5th Edition, 2006.

[20] M. K. Ganai and A. Gupta. Accelerating high-level bounded model checking. In *ICCAD*, pp. 794–801, 2006.

[21] F. Ivancic et al. Model Checking C programs using F-Soft. In *ICCD*. pp. 297–308, 2005.

[22] N. Joseph and K. Hee. Basic Posets. World Scientific Pub Co Inc, First Edition, 1999.

[23] J. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*. North-Holland, pp. 21–28, 1962.

[24] F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In *VSTTE*, pp. 146–161, 2012.

[25] C. Pasareanu and W. Visser, Verification of Java Programs Using Symbolic Execution and Invariant Generation. In *SPIN*, LNCS 2989, pp. 164–181, 2004.

[26] W. Visser and P. Mehlitz Model Checking Programs with Java PathFinder. In *SPIN*, LNCS 3639, pp. 27, 2005.

[27] P. Prabhu et al. Interprocedural Exception Analysis for C++. In *ECOOP*, pp. 583–608. 2011.

[28] R. L. Sites. Some thoughts on proving clean termination of programs. Stanford, CA, USA, Tech. Rep., 1974.

[29] C. Wintersteiger. Compiling GOTO-Programs, http://www.cprover.org/goto-cc/, 2009.

[30] J. Yang et al. Object Model Construction for Inheritance in C++ and its Applications to Program Analysis. In *CC*, LNCS 7210, pp. 144–164, 2012.