Received: 24 May 2021 | Revised: 5 August 2021 | Accepted: 6 August 2021

DOI: 10.1002/stvr.1793

RESEARCH ARTICLE

WILEY

Model checking C++ programs

Felipe R. Monteiro[1] | **Mikhail R. Gadelha**[2] | **Lucas C. Cordeiro**[3]

[1]Federal University of Amazonas, Manaus, Brazil

[2]Igalia, A Coruña, Spain

[3]University of Manchester, Manchester, UK

Correspondence
Lucas C. Cordeiro, University of Manchester, Manchester, UK.
Email: lucas.cordeiro@manchester.ac.uk

Funding information
Engineering and Physical Sciences Research Council; Nokia Institute of Technology; UK Research and Innovation

Summary
In the last three decades, memory safety issues in system programming languages such as C or C++ have been one of the most significant sources of security vulnerabilities. However, there exist only a few attempts with limited success to cope with the complexity of C++ program verification. We describe and evaluate a novel verification approach based on bounded model checking (BMC) and satisfiability modulo theories (SMT) to verify C++ programs. Our verification approach analyses bounded C++ programs by encoding into SMT various sophisticated features that the C++ programming language offers, such as templates, inheritance, polymorphism, exception handling, and the Standard Template Libraries. We formalize these features within our formal verification framework using a decidable fragment of first-order logic and then show how state-of-the-art SMT solvers can efficiently handle that. We implemented our verification approach on top of ESBMC. We compare ESBMC to LLBMC and DIVINE, which are state-of-the-art verifiers to check C++ programs directly from the LLVM bitcode. Experimental results show that ESBMC can handle a wide range of C++ programs, presenting a higher number of correct verification results. Additionally, ESBMC has been applied to a commercial C++ application in the telecommunication domain and successfully detected arithmetic-overflow errors, which could potentially lead to security vulnerabilities.

KEYWORDS
C++, memory safety, model checking, SMT, software verification

1 | INTRODUCTION

Software verification plays an essential role in ensuring overall product reliability as security becomes a major concern [1]. For more than 30 years now, memory safety issues in system programming languages such as C or C++ have been among the major sources of security vulnerabilities [2]. For instance, the Microsoft Security Response Center reported that approximately 70% of their security issues every year are due to memory-safety violations in their C and C++ code [3]. Beyond memory safety, undefined behaviour (e.g., signed-integer overflow) also represents another crucial source of errors that could potentially lead to security vulnerabilities [4].

Over the last 15 years, formal techniques dramatically evolved [5], its adoption in industry has been growing [6-9], and several tools to formally verify C programs have been proposed [10]. However, there exist only a few attempts with limited success to cope with the complexity of C++ program verification [11-18]. The main challenge here is to support sophisticated features that the C++ programming language offers, such as templates, sequential and associative template-based containers, strings & streams, inheritance, polymorphism, and exception handling. Simultaneously, to be attractive for mainstream software development, C++ verifiers must handle large programs, maintain high speed and soundness, and support legacy designs.

In an attempt to cope with ever-growing system complexity, bounded model checking (BMC) based on satisfiability modulo theories (SMT) has been introduced as a complementary technique to Boolean satisfiability (SAT) for alleviating the state explosion problem [19]. In this paper, we describe and evaluate a novel SMT-based BMC approach to verify C++ programs integrated into ESBMC [20-23], a state-of-the-art context-bounded model checker. ESBMC can check for undefined behaviours and memory safety issues such as under- and overflow arithmetic, division-by-zero, pointer safety, array out-of-bounds violations, and user-defined assertions.

Our major contributions are twofold: (i) we present a C++ operational model, an abstract representation of the Standard Template Libraries (STL) that reflects their semantics and enables ESBMC to verify specific properties related to C++ structures (e.g., functional properties of standard containers) via function contracts (i.e., pre- and post-conditions), in addition to memory safety properties; (ii) we also describe and evaluate novel approaches to handle exceptions in C++ programs (e.g., exception specification for functions and methods), which previous approaches could not handle [12,14,15]. We also present an overview of ESBMC's type-checking engine and how it handles templates, inheritance, and polymorphism. Finally, we compare our approach against LLBMC [12], a state-of-the-art bounded model checker based on SMT solvers, and DIVINE [16], a state-of-the-art explicit-state model checker, both for C and C++ programs. Our experimental evaluation contains a broad set of benchmarks with over 1500 instances, where ESBMC reaches a success rate of 84.27%, outperforming LLBMC and DIVINE.

This article is a substantially revised and extended version of a previous contribution by Ramalho et al. [24]. The major differences here are (i) we extend the C++ operational model structure to handle new features from the STL (e.g., associative template-based containers); (ii) we provide details about the C++ rules used to throw and catch exceptions; (iii) we support `terminate` and `unexpected` handlers; and (iv) we extend approximately 36% of our experimental evaluation with a completely new set of benchmarks.

The remainder of this article is organized as follows. Section 2 gives a brief introduction to BMC and describes the background theories of the SMT solvers relevant to our contributions. In Section 3, we describe the aspects of C++ handled in type-checking; that is, our current approach to support templates and the mechanism to support inheritance and polymorphism. We then present the main contributions, Section 4 presents the operational model to replace the STL in the verification process; and Section 5 describes the exception handling encoding. Section 6 presents the results of our experimental evaluation. Finally, in Section 7, we discuss the related work, and we conclude in Section 8 along with our future research directions.

2 | BACKGROUND THEORY

ESBMC is a bounded model checker based on CProver framework [25] aimed to support SMT solvers natively. ESBMC generates verification conditions (VCs) for a given C or C++ program and encodes them using different SMT background theories (i.e., linear-integer and real arithmetic and bit-vectors) and solvers (i.e., Boolector [26], Z3 [27], Yices [28], MathSAT [29], and CVC4 [30]). ESBMC represents one of the most prominent BMC tools for software verification, according to the last editions of the Intl. Competition on Software Verification (SV-COMP) [31] and the Intl. Competition on Software Testing [32]; in particular, it was ranked at the top three verifiers in the overall ranking of SV-COMP 2020 [31]. ESBMC has been applied to verify (embedded) software in digital filters [33] and digital controllers [34], and unmanned aerial vehicles [35].

2.1 | Bounded model checking

In BMC, the program to be analysed is modelled as a state transition system, which is extracted from the control-flow graph (CFG) [36]. This graph is built as part of a translation process from program code to static single assignment (SSA) form. A node in the CFG represents either a (non-) deterministic assignment or a conditional statement, while an edge in the CFG represents a possible change in the program's control location.

Given a transition system M, a property ϕ, and a bound k, BMC unrolls the system k times and translates it into a VC ψ, such that ψ is satisfiable if and only if ϕ has a counterexample of length k or less [19]. The associated model checking problem is formulated by constructing the following logical formula:

$$\psi_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg\phi(s_i), \tag{1}$$

given that ϕ is a safety property, I is the set of initial states of M and $T(s_i, s_{i+1})$ is the transition relation of M between steps i and $i+1$. Hence, $I(s_0) \wedge \bigwedge_{i=0}^{j-1} T(s_i, s_{i+1})$ represents the executions of M of length j and the formula (1) can be

satisfied if and only if, for some $j \leq k$, there exists a reachable state at step j in which ϕ is violated. If the formula (1) is satisfiable, then the SMT solver provides a satisfying assignment, from which we can extract the values of the program variables to construct a counterexample. A counterexample for a property ϕ is a sequence of states $s_0, s_1, \ldots, s_k$ with $s_0 \in S_0$ and $T(s_i, s_{i+1})$ with $0 \leq i < k$.

If the formula (1) is unsatisfiable, we can conclude that no error state is reachable in k steps or less. In this case, BMC techniques are not complete because there might still be a counterexample that is longer than k. Completeness can only be ensured if we know an upper bound on the depth of the state space. This means that if we can ensure that we have already explored all the relevant behaviour of the system, and searching any deeper only exhibits states that have already been verified [37].

2.2 | Satisfiability modulo theories

SMT decides the satisfiability of a fragment of quantifier-free first-order formulae using a combination of different background theories. It 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 τ and a quantifier-free formula ψ, we say that ψ is τ-satisfiable if and only if there exists a structure that satisfies both the formula and the sentences of τ, or equivalently if $\tau \cup \{\psi\}$ is satisfiable [38]. Given a set $\Gamma \cup \{\psi\}$ of formulae over τ, we say that ψ is a τ-consequence of Γ, and write $\Gamma \vDash_\tau \psi$, if and only if every model of $\tau \cup \Gamma$ is also a model of ψ. Checking $\Gamma \vDash_\tau \psi$ can be reduced in the usual way to checking the τ-satisfiability of $\Gamma \cup \{\neg\, \psi\}$.

ESBMC heavily uses the (non-extensional) theory of arrays $\mathcal{T}_\mathcal{A}$ based on the McCarthy axioms [39], to properly encode properties and behaviours of the STL models (*cf.* Section 4) and the C++ exception handling features (cf. Section 5). We define conditional expressions [40] over bitvectors using the $ite(c, t_1, t_2)$ operator, where c is the condition expression, t_1 is the consequent branch $ite(\top, t_1, t_2) = t1$, and t_2 is the alternative branch $ite(\bot, t_1, t_2) = t_2$. The operation $select(a, i)$ 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 [27,30,41]:

$$\begin{aligned} i = j &\Rightarrow select(store(a,i,v),j) = v \\ \neg\,(i = j) &\Rightarrow select(store(a,i,v),j) = select(a,j) \end{aligned}$$

Finally, an important component of our models is the *memcpy pattern* through lambda terms introduced by Preiner, Niemetz, and Biere [40]. It allows us to reason about operations over multiple indices without the need for quantifiers. Here, the $memcpy(a, b, i, k, n)$ operation denotes a copy of n elements from array a starting at position i to array b at the position k.

3 | STATIC TYPE CHECKING OF C++ PROGRAMS

The first steps when verifying C++ programs are the source-code parser and the type-checker, which are language-specific in ESBMC (see Figure 1). For C++, the parser is heavily based on the GNU C++ Compiler (GCC) [42], which allows ESBMC to find and report most of the syntax errors already reported by GCC. Type-checking provides all information used by the model; thus, a better type-checker means it is possible to model more programs. The code is statically analysed on type-checking, including assignment checks, type-cast checks, pointer initialization checks, and function call checks. Furthermore, ESBMC handles three major C++ features on type-checking: template instantiation (i.e., after type-checking, all referenced templates are instantiated with concrete types), compile-time and runtime polymorphism, and inheritance (i.e., it replicates the methods and attributes of the base classes to the inherited class, which will have direct access).

By the end of the type-check, the Intermediate Representation (IR) creation is completed and used by the GOTO converter to generate the GOTO program. The verification of C programs is slightly different as it uses clang as a front-end to parse and type-check the program, as described in our previous work [22,23]; the output, however, it is the same: a type-checked IR.

The GOTO converter converts the type-checked IR into GOTO expressions; this conversion simplifies the IR of the original program (e.g., replacing of `switch` and `while` by `if` and `goto` statements). The symbolic engine converts the GOTO program into SSA form by unrolling loops up to bound k. Assertions are inserted into the resulting SSA expressions to verify memory-safety properties (e.g., array out-of-bounds access, arithmetic under- and overflow,

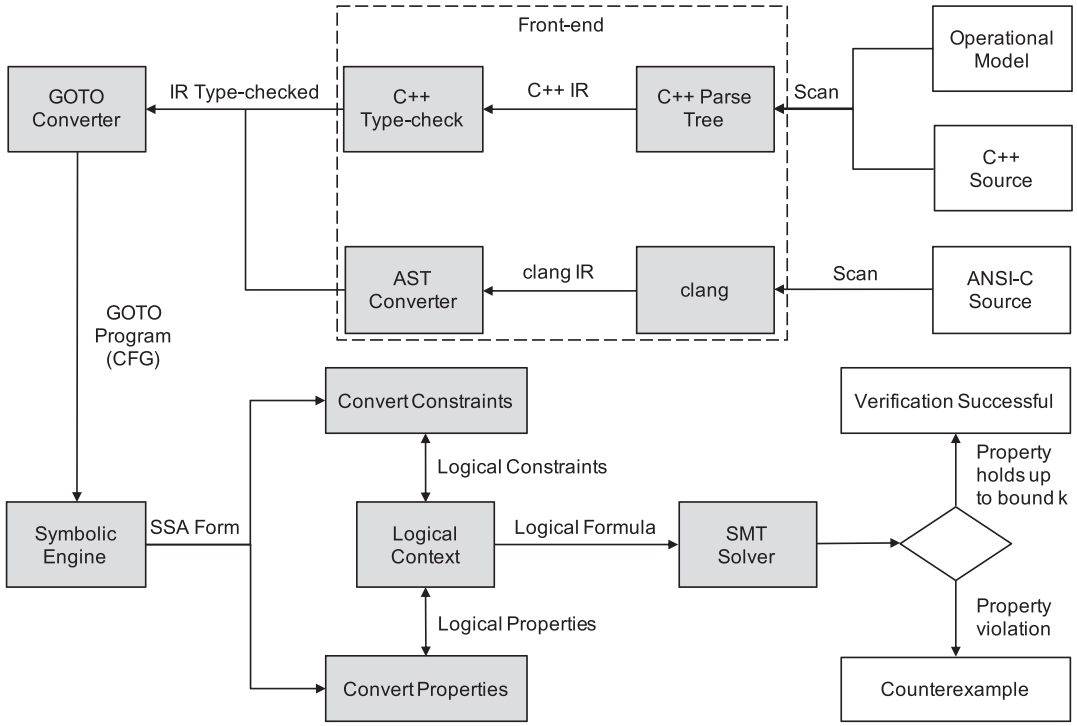


FIGURE 1 ESBMC architectural overview. White rectangles represent input and output; grey rectangles represent the steps of the verification

memory leaks, double frees, division-by-zero, etc.). Also, most of the exception handling is carried out in this step, such as the search for valid `catch`, assignment of a thrown object to a valid `catch` object, replacement of `throw` statements by GOTO expressions and exception specs for function calls (cf. Section 5).

Finally, two sets 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. The two sets of formulae will be used as input for an SMT solver that will produce a counterexample if there exists a violation of a given property, or an unsatisfiable answer if the property holds.

3.1 | Template instantiation

Templates are not runtime objects [43]. When a C++ program is compiled, classes and functions are generated from templates. Those templates are removed from the final executable. ESBMC has a similar process in which templates are only used until the type-checking phase, where all templates are instantiated and the classes and functions are generated. Any instantiated functions and classes are no longer templates. Hence, at the end of the type-checking phase, all templates are completely discarded. In ESBMC, the entire verification process of C++ programs, which make use of templates, is essentially split into two steps: *creation of templates* and *template instantiation*. The *creation of templates* is straightforward. It happens during the parsing step when all generic data types of the generated C++ IR are properly marked as `generic` and each specialization is paired with its corresponding primary template. No instantiated function or class is created during parsing because ESBMC does not know which template types will be instantiated.

A template instantiation happens when a template is used, instantiated with data types (e.g., `int`, `float`, or `string`). ESBMC performs an in-depth search in the C++ IR during the type-checking process to trigger all instantiations. When a template instantiation is found, ESBMC firstly identifies which type of template it is dealing with (i.e., either `class` or `function` template) and which template arguments are used. It then searches whether an IR of that type was already created, i.e., whether its arguments have been previously instantiated. If so, no new IR is created; this avoids duplicating the IR, thus reducing the memory requirements of ESBMC. If there is no IR of that type, a new IR is created, used in the instantiation process, and saved for future searches. To create a new IR, ESBMC must select the most specialized template for the set of template arguments; therefore, ESBMC performs another search in the IR to select the proper template definition. ESBMC then checks whether there is a (partial or explicit) template specialization, matching the set of data types in the instantiation. If ESBMC does not find any template specialization, which

```
1  #include<cassert>
2  using namespace std;
3
4  // template creation
5  template <typename T>
6  bool qCompare(const T a, const T b) {
7    return (a > b) ? true : false;
8  }
9
10 template <typename T>
11 bool qCompare(T a, T b) {
12   return (a > b) ? true : false;
13 }
14
15 // template specialization
16 template<>
17 bool qCompare(float a, float b) {
18   return (b > a) ? true : false;
19 }
20
21 int main() {
22   // template instantiation
23   assert((qCompare(1.5f, 2.5f)));
24   assert((qCompare<int>(1, 2) == false));
25   return 0;
26 }
```

FIGURE 2 Function template example

matches the template arguments, it will select the primary template definition. Once the most specialized template is selected, ESBMC performs a transformation to replace all generic types for the data types specified in the instantiation; this transformation is necessary because, as stated previously, at the end of the C++ type-checking phase, all templates are removed.

In order to concretely demonstrate the instantiation process in ESBMC, Figure 2 illustrates an example of function templates usage, which is based on the example **`spec29`** extracted from the GCC test suite.[1] The first step, the template creation, happens when the declaration of a template function (lines 5–19) is parsed. At this point, the generic IR of the template is created with a generic type. The second step, template instantiation, happens when the template is used. In Figure 2, the template is instantiated twice (lines 23 and 24). It is also possible to determine the type implicitly (line 23) or explicitly (line 24). In implicit instantiation, the data type is determined by the types of the used parameters. In contrast, in the explicit instantiation, the data type is determined by the value passed between the < and > symbols.

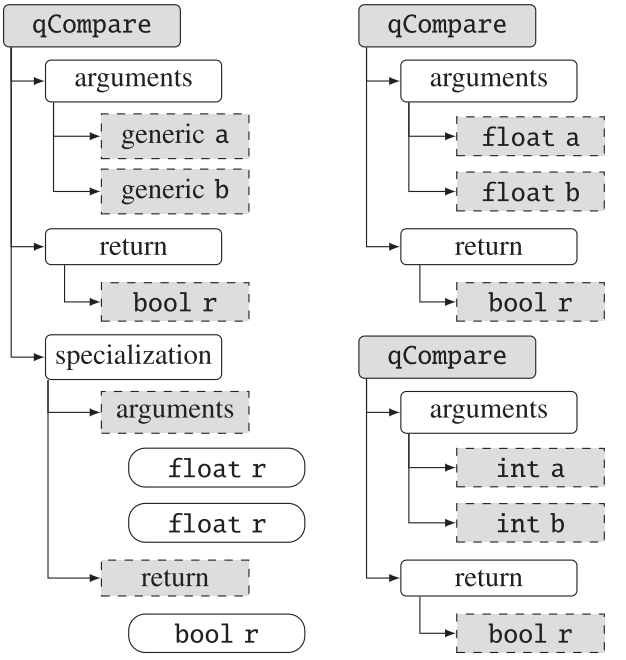
Figure 3 illustrates the generic IR and the instantiated IRs generated from the code in Figure 2. Figure 3a illustrates the generic IR generated from the **`qCompare`** function template and its specialization, while Figure 3b shows the IRs created from instantiating this template with data type **`float`** (line 23) and **`int`** (line 24). The function body is omitted in this figure, but it follows the same instantiation pattern. The generic IR is built with the function name, which is used as a key for future searches, the IR's arguments and return type, as can be seen in Figure 3a. Note that the data type is labelled as **`generic`**, which means that the type is generic. In Figure 3b, the data types that were previously labelled as **`generic`** are now labelled as **`float`** for the first instantiation and **`int`** for the second instantiation, which means that these instantiated IRs are not templates anymore and will not be removed at the end of the type-check phase. Finally, as described earlier, at the end of the type-check phase, the generic IR illustrated in Figure 3a is discarded.

After the template instantiation, the verification process resumes, as described by Cordeiro et al. [44]. ESBMC is currently able to handle the verification of C++ programs with template functions, class templates, and (partial and explicit) template specialization, according to the C++03 standard [45]. The implementation of template instantiation in ESBMC is based on the formalization previously presented by Siek and Taha [46] who introduced the first proof of type safety of the template instantiation process for C++03 programs.

3.2 | Inheritance

In contrast to Java, which only allows single inheritance, where derived classes have only one base class, C++ also allows multiple inheritances, where a class may inherit from one or more unrelated base classes [47]. This particular

[1] https://github.com/nds32/gcc/



(a) Generic IR generated from the `qCompare` function template with generic type in Fig. 2.

(b) Instantiated IRs generated from the template instantiation with types `float` and `int` in Fig. 2.

FIGURE 3 Example of IR creation

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 [48,49]. Multiple inheritance in C++ includes features that raise exciting challenges for model checking such as repeated and shared inheritance of base classes, object identity distinction, and dynamic dispatch [50].

In ESBMC, inheritance is handled by replicating the methods and attributes of the base classes to the derived class, obeying the rules of inheritance defined in the C++03 standard [45]. In particular, we follow these specifications to handle multiple inheritance and avoid issues such as name clashing when replicating the methods and attributes. For example, if two or more base classes implement a method that is not overridden by the derived class, every call to this method must specify which "version" inherited it is referring to. The rules are checked in the type-check step of the verification (cf. Section 3).

A formal description to represent the relationship between classes can be described by a class hierarchy graph. 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 (i.e., if there exists a path from class X to class Y whose first edge is virtual), and $\prec_r \subseteq C \times C$ are *replicated inheritance* edges (i.e., if a class inherits from a base class that does not contain virtual methods). We also define the set of all inheritance edges $\prec_{sr} = \prec_s \cup \prec_r$. Thus, $(C, \preceq_{sr})$ is a partially ordered set [51] and $\preceq_{sr}$ is anti-symmetric (i.e., if one element A of the set precedes B, the opposite relation cannot exist). Importantly, during the replication process of all methods and attributes from the base classes to the derived ones, the inheritance model considers the access specifiers related to each component (i.e., **`public`**, **`protected`**, and **`private`**) and its friendship [47]; therefore, we define two rules to deal with such restrictions: (i) only **`public`** and **`protected`** class members from base classes are joined in the derived class and (ii) if class $X \in C$ is a friend of class $Y \in C$, all private members in class X are joined in class Y.

As an example, Figure 4 shows an UML diagram that represents the **`Vehicle`** class hierarchy, which contains multiple inheritance. The replicated inheritance in the **`JetCar`** class relation can be formalized by $\langle C, \emptyset, \{$(**`JetCar`**, **`Car`**), (**`JetCar`**, **`Jet`**)$\}\rangle$.

ESBMC 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. 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. However, we replicate information to any new class, thus wasting memory resources.

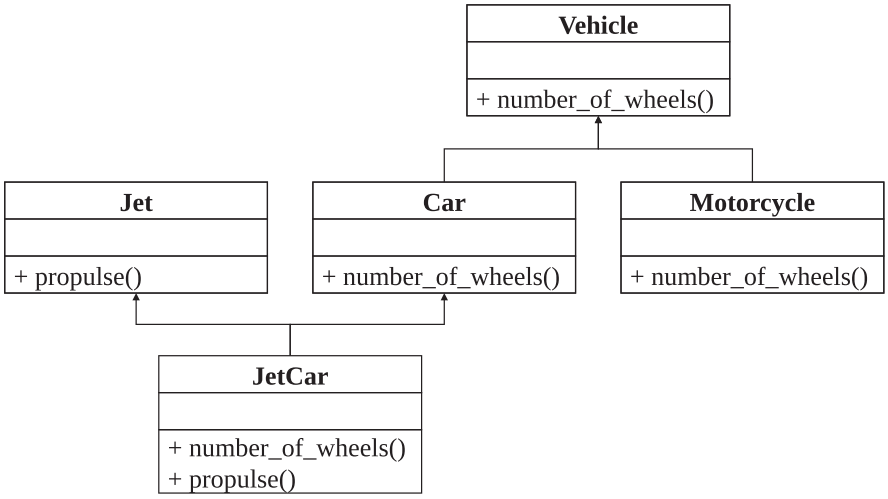


FIGURE 4 `Vehicle` class hierarchy UML diagram

In addition, we also support *indirect inheritance*, where a class inherits features from a derived class with one or more classes not directly connected. *Indirect inheritance* is automatically handled due to our replication method: any derived class will already contain all methods and attributes from their base classes, which will be replicated to any class that derives from them. In Figure 4, we have `JetCar`$\leq_{sr}$`Car` and `Car`$\leq_{sr}$`Vehicle`. Thus, the `JetCar` class can access features from the `Vehicle` class, but they are not directly connected.

In object-oriented programming, the use of *shared inheritance* is very common [47]. In contrast to other approaches (e.g., the one proposed by Blanc, Groce, and Kroening [14]), ESBMC is able to verify this kind of inheritance. A pure virtual class does not implement any method and, if an object tries to create an instance of a pure virtual class, ESBMC will fail with a `CONVERSION ERROR` message (since it is statically checked during type-checking).

3.3 | Polymorphism

In order to handle polymorphism, that is, allowing variable instances to be bound to references of different types, related by inheritance [52], ESBMC implements a virtual function table (i.e., `vtable`) mechanism [53]. When a class defines a virtual method, ESBMC creates a `vtable`, which contains a pointer to each virtual method in the class. If a derived class does not override a virtual method, then the pointers are copied to the virtual table of the derived class. In contrast, if a derived class overrides a virtual method, then the pointers in the virtual table of the derived class will point to the overridden method implementation. Whenever a virtual method is called, ESBMC executes the method pointed in the virtual table. ESBMC also supports the unary scope resolution operator (i.e., `::`), which, in this context, enables a derived class to access members from its parents, a key component to support multiple inheritance.

Consider the program in Figure 5, which contains a simplified version of the class hierarchy presented in Figure 4. In the program, a class `Vehicle` is base for two classes, `Motorcycle` and `Car`. The class `Vehicle` defines a pure virtual method `number_of_wheel()`, and both classes `Motorcycle` and `Car` implement the method, returning 2 and 4, respectively. The program creates an instance of `Motorcycle` or `Car`, depending on a nondeterministic choice, and assigns the instance to a `Vehicle` pointer object `v`. Finally, through the polymorphic object `v`, the program calls `number_of_wheel()` and checks the returned value. We omit a call to `delete` (that would free the pointer `v`) to simplify the GOTO instructions.

Figure 6a shows the GOTO program (resulted from the type-checking phase) generated for the program in Figure 5. Note that, when building the polymorphic object `v`, the `vtable`'s pointer for the method `number_of_wheel()` is first assigned with a pointer to the method `number_of_wheel()` in class `Vehicle` (see lines 10 and 17 in Figure 6a); this happens because the constructor for both `Car` and `Motorcycle` first call the base constructor in the original program (see lines 13 and 20 in Figure 5). They are then assigned the correct method address (see lines 12 and 19 in Figure 6a) in the constructors of the derived classes, that is, `Motorcycle` and `Car`, respectively.

In the SSA form shown in Figure 6b, every branch creates a separate variable, which are then combined when the control-flow merges. In Figure 6b, we generate two branches (i.e., `v1` and `v2`) and a ϕ-node (i.e., `v3`) to merge both

```
1  #include <cassert>
2
3  class Vehicle
4  {
5  public:
6    Vehicle() {};
7    virtual int number_of_wheels() = 0;
8  };
9
10 class Motorcycle : public Vehicle
11 {
12 public:
13   Motorcycle() : Vehicle() {};
14   virtual int number_of_wheels() { return 2; };
15 };
16
17 class Car : public Vehicle
18 {
19 public:
20   Car() : Vehicle() {};
21   virtual int number_of_wheels() { return 4; };
22 };
23
24 int main()
25 {
26   bool foo = nondet();
27
28   Vehicle* v;
29   if(foo)
30     v = new Motorcycle();
31   else
32     v = new Car();
33
34   bool res;
35   if(foo)
36     res = (v->number_of_wheels() == 2);
37   else
38     res = (v->number_of_wheels() == 4);
39   assert(res);
40   return 0;
41 }
```

FIGURE 5 C++ program using a simplified version of the UML diagram in Figure 4. The program nondeterministically cast the derived class to a base class. The goal is to check if the correct **`number_of_wheels()`** is called, from the base class

branches. For instance, the variable **`v1`** represents the branch, where the polymorphic variable **`v`** gets assigned an object of type **`Motorcycle`**, while **`v2`** represents the branch, where **`v`** gets assigned an object of type **`Car`**. They are then merged into **`v3`**, depending on the initial nondeterministic choice (see line 13 in Figure 6b). There exists no side-effect in the SSA form, as it can use the correct definition of **`number_of_wheels()`** in the ϕ-node. The type-checker does all the heavy lifting.

4 | C++ OPERATIONAL MODEL

The C++ programming language offers a collection of libraries, called STL, to provide most of the functionalities required by a programmer [45]. However, the direct inclusion of the STL into the verification process overcomplicates the analysis of C++ programs, as it contains code fragments not relevant for verification (e.g., optimized assembly code) [18,24]. Its implementation is based on a pointer structure that degrades the verification performance [14]. In particular, existing BMC tools adopt two different memory models: a *fully byte-precise* [12] or an *object-based* [54,55] memory model. Note that BMC tools reduce bounded program traces to a decidable fragment of first-order logic, which requires us to eliminate pointers in the model checker. They use static analysis to approximate each pointer variable the set of data objects (i.e., memory chunks) at which it might point at some stage in the program execution. For a *fully byte-precise* memory model, BMC tools treat all memory as a single byte array, upon which all pointer accesses are decomposed into byte operations. This can lead to performance problems due to the repeated updates to the memory array that needs to be reflected in the SMT formula. For an *object-based* memory model, this approach's performance suffers if pointer offsets cannot be statically determined, for example, if a program reads a byte from an arbitrary offset into a structure. The resulting SMT formula is large and unwieldy, and its construction is error-prone.

To reduce verification complexity, ESBMC uses an abstract representation of the STL, called the C++ Operational Model (COM), which adds function contracts [56] (i.e., pre- and post-conditions) to all STL function/method calls. Thus, all those function contracts are verified by ESBMC. The purpose of the verification is to check whether a given

```
main() (c::main):
   FUNCTION_CALL:
      return_value_nondet$1=nondet()
   bool foo;
   foo = return_value_nondet$1;
   class Vehicle * v;
   IF !foo THEN GOTO 1
   new_value1 = new class Motorcycle;
   new_value1->vtable->number_of_wheels =
     &Vehicle::number_of_wheel();
   new_value1->vtable->number_of_wheels =
     &Motorcycle::number_of_wheel();
   v = (class Vehicle *)new_value;
   GOTO 2
1: new_value2 = new class Car;
   new_value2->vtable->number_of_wheels =
     &Vehicle::number_of_wheel();
   new_value2->vtable->number_of_wheels =
     &Car::number_of_wheel();
   v = (class Vehicle *)new_value;
   bool res;
2: IF !foo THEN GOTO 3
   FUNCTION_CALL:
      return_value_number_of_wheels =
         *v->vtable->number_of_wheel()
   res = wheels == 2
   GOTO 4
3: FUNCTION_CALL:
      return_value_number_of_wheels =
         *v->vtable->number_of_wheel()
   res = wheels == 4
4: ASERT res
   RETURN: 0
END_FUNCTION
```

(a) GOTO instructions.

```
return_value_nondet1 = nondet_symbol(symex::0)
foo1 = return_value_nondet1
new_value11 = new_value10
  WITH [vtable = new_value10.vtable
    WITH [number_of_wheel =
      &Motorcycle::number_of_wheels()]]
v1 = new_value11
new_value12 = new_value10
  WITH [vtable = new_value10.vtable
    WITH [number_of_wheel =
      &Car::number_of_wheels()]]
v2 = new_value12
v3 = (foo1 ? v1 : v2);
return_value_number_of_wheels1 = 2
res1 = (return_value_number_of_wheels1 == 2)
return_value_number_of_wheels2 = 4
res2 = (return_value_number_of_wheels2 == 4)
res3 = (foo1 ? res1 : res2)
```

(b) SSA form.

FIGURE 6 Internal representations of the program in Figure 5

```
template<typename T>
static T get_from_vector(
          const std::vector<uint8_t>& vec,
          const size_t current_index)
{
    T result;
    uint8_t *ptr = (uint8_t *) &result;
    size_t idx = current_index + sizeof(T);
    while(idx > current_index)
        *ptr++ = vec[--idx];
    return result;
}
```

(a) Code snippet.

```
reference vector::operator[](size_type i)
{
    __ESBMC_HIDE:;
    __ESBMC_assert(i >= 0 && i < _size,
                   "Out of bounds violation");
    return buf[i];
}
```

(b) Operational model for `vector::operator[]`.

FIGURE 7 Example from Stack Overflow (best accepted answer) that contains improper input validation (CWE-20) and out-of-bounds read (CWE-125) vulnerabilities

program uses STL correctly without hitting a bogus state (e.g., calling **`vector::operator[]`** with an out-of-range parameter leads to undefined behaviour). A similar technique, proposed by Blanc et al. [14], has been used to verify pre-conditions on programs. However, ESBMC extends that approach by also checking the post-conditions, which improves its effectiveness, as shown in our experimental evaluation (cf. Section 6).

Figure 7a shows a code snippet considered as the best-accepted answer for a Stack Overflow question.[2] Nevertheless, line 10 could lead to an out-of-bound violation (CWE-125 vulnerability) [57]. ESBMC detects the erroneous state through the operational model for **`vector::operator[]`** (see Figure 7b), which contains an assertion to check for out-of-bound accesses. The model also keeps track of the values stored in the container using a buffer (**`buf`**), so it also guarantees the post-condition for the operator, that is, return a reference to the element at specified location **`i`**.

[2]Available at https://stackoverflow.com/questions/41028862.

TABLE 1 Overview of the C++ operational model

Standard C++03 libraries—operational model							
C standard libraries	**General**	**Streams input/output**	**Containers**	**Language support**	**Numeric**	**Strings**	**Localization**
`cassert`	`memory`	`ios`	`bitset`	`exception`	`complex`	`string`	`locale`
`cctype`	`stdexcept`	`iomanip`	`deque`	`limits`	`random`		
`cerrno`	`utility`	`iosfwd`	`list`	`new`	`valarray`		
`cfloat`	`functional`	`iostream`	`map`	`typeinfo`	`numeric`		
`ciso646`		`istream`	`multimap`				
`climits`		`ostream`	`set`				
`clocale`		`streambuf`	`multiset`				
`cmath`		`sstream`	`vector`				
`complex`		`fstream`	`stack`				
`csetjmp`			`queue`				
`csignal`			`algorithm`				
`cstdarg`			`iterator`				
`cstddef`							
`cstdio`							
`cstdlib`							
`cstring`							
`ctime`							

Our COM mimics the structure of the STL, as shown in Table 1. All ANSI-C libraries are natively supported by ESBMC, as described by Cordeiro et al. [20]. For all libraries under categories **`General`**, **`Language Support`**, **`Numeric`**, and **`Localization`**, COM adds pre-conditions extracted directly from documentation [45], specifically designed to detect memory-safety violations (e.g., nullness and out-of-bounds checks).

One of the challenges of modelling COM is the support for containers, strings, and streams, which requires the injection of pre- and post-conditions to check for functional properties correctly, as shown in the example illustrated in Figure 7b (cf. the pre-conditions in lines 4–5). In this specific example, we check the **`vector`** upper and lower bounds before retrieving its content to detect an out-of-bounds read in line 10 of Figure 7a. COM models sequential and associative containers along with their iterators. In particular, libraries **`list`**, **`bitset`**, **`deque`**, **`vector`**, **`stack`**, and **`queue`** belong to the sequential group, while libraries **`map`**, **`multimap`**, **`set`**, and **`multiset`** belong to the associative group. COM models strings and streams objects as arrays of bytes to properly encode them using the theory of arrays (cf. Section 2.2); therefore, **`string`** and all Stream I/O libraries also belong to the sequential group.

4.1 | Core language

The gist of COM enables ESBMC to encode features of standard containers, strings, and streams using the theory of arrays $\mathcal{T}_A$. To properly formalize the verification of our model, we extend the previous core container language presented by Ramalho et al. [24] to include a representation for keys, which allows us to reason about associative containers as well. The core language defines the syntactic domains values V, keys K, iterators I, pointers P, container C and integers $\mathbb{N}$ as follows,

$$
\begin{aligned}
V &:= v \mid *i_v \\
K &:= k \mid *i_k \\
I &:= i \mid C.insert(I,V) \mid C.insert(K,V) \\
&\quad\; C.search(K) \mid C.search(V) \\
&\quad\; C.erase(I) \\
P &:= p \mid P(+ \mid -)P \mid c_v \mid c_k \mid i_v \mid i_k \\
C &:= c \\
\mathbb{N} &:= n \mid \mathbb{N}(+ \mid *|\ldots)\mathbb{N} \mid size \mid pos
\end{aligned}
$$

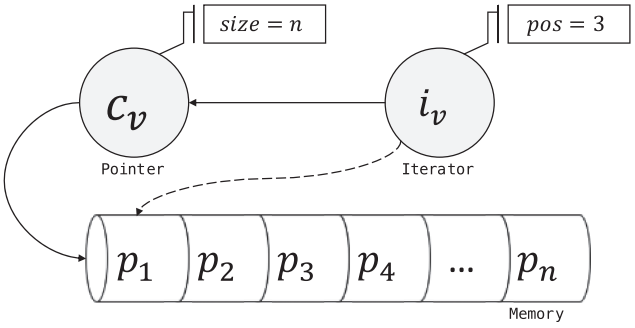


FIGURE 8 Abstraction for sequential containers

Here v, k, p, i, c and n are classes of variables of type V, K, P, I, C and $\mathbb{N}$, respectively. For iterators, we use the notation $*i_v$ to denote the value stored in the memory location i_v. Based on such domains, we also define $P(+ \mid -)P$ as valid pointer operations and $\mathbb{N}(+ \mid *|\ldots)\mathbb{N}$ as valid integer operations. Each operation shown in the core container syntax (e.g., $C.insert(I, V)$) is explained in Sections 4.2 and 4.3.

All methods from the sequential and associative groups can be expressed as combinations/variations of three main operations: insertion ($C.insert(I, V)$), deletion ($C.erase(I)$), and search ($C.search(V)$). Each operation is described in our model as a Hoare triple $\{\mathcal{P}\}\ C\ \{\mathcal{Q}\}$ that represents the function contract scheme implemented by COM. Normally all side-effects would be stated in the post-condition $\mathcal{Q}$ for verification. However, as part of the SSA transformation, side effects on iterators and containers are made explicit. Operations return new iterators and containers with the same contents, except for the fields that have just been updated. Thus, the translation function $\mathcal{C}$ contains primed variables (e.g., c' and i') to represent the state of model variables after the respective operation. Finally, all models take advantage of *memcpy pattern* through lambda terms [40], which enables us to describe array operations over multiple indices in a clear and concise manner (cf. Section 2.2).

4.2 | Sequential containers

Sequential containers are built into a structure to store elements with a sequential order [47]. In our model, a sequential container c consists of a pointer c_v that points to a valid memory location and an integer *size* that stores the number of elements in the container. Similarly, an iterator i is modelled using two variables: an integer i_{pos}, which contains the index value of the container pointed by the iterator and a pointer i_v, which points to the memory location referred by the iterator. In our model, the defined notation $*i$ is equivalent to $select(i_v, i_{pos})$. Figure 8 gives an overview of our abstraction for all sequential containers.

The statement $c.insert(i, v)$ becomes $(c', i') = c.insert(i, v)$ increases the container size, move all elements from position $i.pos$ one memory unit forward, and then insert v into the specified position. Therefore,[3]

$$\begin{aligned}\mathcal{C}\ \ &((c', i') = c.insert(i, v)) :=\\ &c'.size = c.size + 1\\ &\wedge\ memcpy(c.c_v,\ c'.c_v,\ i.pos,\ i.pos + 1,\ c.size - i.pos)\\ &\wedge\ store(c'.c_v,\ i.pos,\ v)\end{aligned} \tag{2}$$

that induces the following pre- and post-conditions,

$$\begin{aligned}\mathcal{P}\ \ &((c', i') = c.insert(i, v)) :=\\ &v \neq null\\ &\wedge\ c.c_v \neq null\\ &\wedge\ i.i_v \neq null\\ &\wedge\ 0 \leq i.pos < c'.size\end{aligned} \tag{3}$$

[3]Note that SMT theories only have a single equality predicate (for each sort). However, here we use the notation ":=" to indicate an assignment of nested equality predicates on the right-hand side of the formula.

$$\begin{aligned}\mathcal{Q}\ \ &((c', i') = c.insert(i, v)) := \\ &select(i'.i_v,\ i'.pos)\ =\ v \\ &\wedge\ i'.i_v = c'.c_v \\ &\wedge\ i'.pos = i.pos\end{aligned} \tag{4}$$

where *null* represents an uninitialized pointer/object. Thus, we define as pre-conditions $\mathcal{P}$ that v and i cannot be uninitialized objects as well as $i.pos$ must be within $c'.c_v$ bounds; similarly, we define as post-conditions $\mathcal{Q}$ that v was correctly inserted in the position specified by i as well as $c'.c_v$ and $i'.i_v$ are equivalent, that is, both point to the same memory location. Importantly, we implement the memory model for containers essentially as arrays, therefore, the range to select elements from memory varies from 0 to $c.size - 1$. Furthermore, the main effect of the *insert* method is mainly captured by Equation (2) that describes the contents of the container array $c'.c_v$ after the insertion in terms of update operations to the container array $c.c_v$ before the insertion.

The erase method works similarly to the insert method. It uses iterator positions, integer values, and pointers, but it does not use values since the exclusion is made by a given position, regardless of the value. It also returns an iterator position (i.e., i'), pointing to the position immediately after the erased part of the container [45]. Therefore,

$$\begin{aligned}\mathcal{C}\ \ &((c', i') = c.erase(i)) := \\ &memcpy(c.c_v,\ c'.c_v,\ i.pos + 1,\ i.pos,\ c.size - (i.pos + 1)) \\ &\wedge\ c'.size = c.size - 1 \\ &\wedge\ i'.pos = i.pos + 1\end{aligned} \tag{5}$$

that induces the following pre- and post-conditions,

$$\begin{aligned}\mathcal{P}\ \ &((c', i') = c.erase(i)) := \\ &i.i_v \neq null \\ &\wedge\ c.c_v \neq null \\ &\wedge\ 0 \leq i.pos < c.size \\ &\wedge\ c.size \neq 0 \Rightarrow c.c_v \neq null\end{aligned} \tag{6}$$

$$\begin{aligned}\mathcal{Q}\ \ &((c', i') = c.erase(i)) := \\ &select(c'.c_v,\ i'.pos)\ =\ select(c.c_v,\ i.pos + 1) \\ &\wedge\ i'.i_v = c'.c_v\end{aligned} \tag{7}$$

where we assume as pre-conditions $\mathcal{P}$ that i must be a valid iterator pointing to a position within the bounds of array $c.c_v$ and c must be non-empty; similarly, we assume as post-conditions $\mathcal{Q}$ that i' must point to the element immediately after the erased one and $c'.c_v$ and $i'.i_v$ point to the same memory location. Finally, a container c with a call $c.search(v)$ performs a search for an element $v$in the container. Then, if such an element is found, it returns an iterator that points to the respective element; otherwise, it returns an iterator that points to the position immediately after the last container's element (i.e., $select(c'.c_v,\ c'.size)$). Hence,

$$\begin{aligned}&\mathcal{C}((c', i') = c.search(v)) := \\ &\quad ite(c.size = 0, \\ &\qquad i'.pos = c.size, \\ &\qquad ite(select(c.c_v, 0) = v, \\ &\qquad\quad i'.pos = 0, \\ &\qquad\quad ... \\ &\qquad\quad ite(select(c.c_v, c.size - 1) = v, \\ &\qquad\qquad i'.pos = c.size - 1, \\ &\qquad\qquad i'.pos = c.size)\ ...\ ))\end{aligned} \tag{8}$$

that induces the following pre- and post-conditions,

$$\mathcal{P}((c', i') = c.search(v)) := \\ v \neq null \tag{9}$$

$$\begin{aligned} \mathcal{Q}\ \ & ((c', i') = c.search(v)) := \\ & c'.c_v = c.c_v \\ & \wedge\ c'.size = c.size \\ & \wedge\ i'.i_v \neq c'.c_v \\ & \wedge\ ite(select(i'.i_v,\ i'.pos)\ =\ select(c'.c_v, i'.pos), \\ & \qquad select(i'.i_v,\ i'.pos) = v, \\ & \qquad select(i'.i_v,\ i'.pos) = select(c'.c_v,\ c'.size)) \end{aligned} \tag{10}$$

where we assume as pre-conditions $\mathcal{P}$ that v and c cannot be an uninitialized objects; similarly, we assume as post-conditions $\mathcal{Q}$ that c' is equivalent to its previous state c, $c'.c_v$ and $i'.i_v$ point to the same memory location, and i' must point to the found element or to $select(c'.c_v,\ c'.size)$.

4.3 | Associative containers

Associative containers consist of elements with a key k and a value v, where each value is associated with a unique key. All elements are internally sorted by their keys based on a strict weak ordering rule [45]. In our model, an associative container c consists of a pointer c_v, for the container's values, a pointer c_k, for the container's keys, and an integer *size*, for the container's size. Figure 9 gives an overview of our abstraction for all associative containers. The relationship between c_k and c_v is established by an index; thus, an element in a given position n in c_k (i.e., $select(c.c_k,\ n)$) is the key associated with the value in the same position n in c_v (i.e., $select(c.c_v,\ n)$). Similarly, iterators for associative containers consist of a pointer i_k that points to the same memory location as c_k, a pointer i_v that points to the same memory location as c_v, and an integer i_{pos} that indexes both i_k and i_v. All operations for associative containers can be expressed as a simplified variation of the three main ones, that is, insertion ($C.insert(K,\ V)$), deletion ($C.erase(I)$), and search ($C.search(K)$).

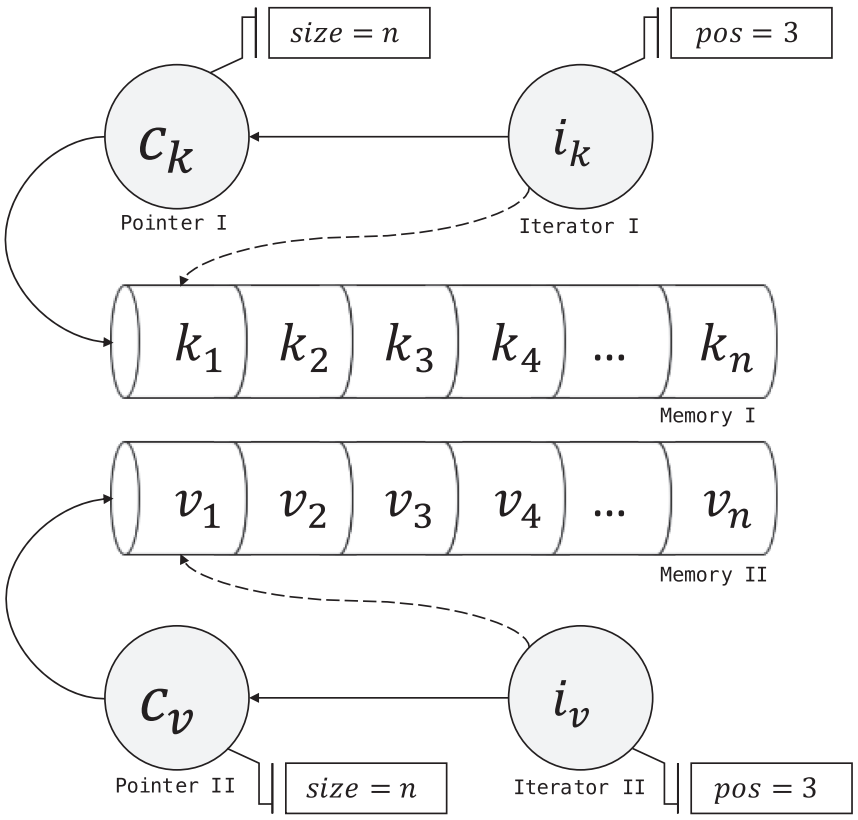


FIGURE 9 Abstraction for associative containers

The order of keys matters in the insertion operation for associative containers. Therefore, given a container c, the method calls $c.insert(k, v)$ inserts the value v associated with the key k into the right order (i.e., obeying a strict weak ordering rule). Here, we use the operator $\prec$ to represent precedence; thus, $x \prec y$ means x precedes y. The insertion returns an iterator that points to the inserted position. However, if k exists, the insertion is not performed and the method returns an iterator that points to the existing element. We checked for three cases, which correspond to each *ite* condition: (i) the empty case first, then (ii) we check whether each position contains a corresponding key or (iii) if we should insert the value based on its precedence. Thus,

$$
\begin{aligned}
&\mathcal{C}((c', i') = c.insert(k, v)) := \\
&\quad ite(c.size = 0, \\
&\qquad i'.pos = 0 \\
&\qquad \wedge\ store(c'.c_k,\ i'.pos,\ k) \\
&\qquad \wedge\ store(c'.c_v,\ i'.pos,\ v) \\
&\qquad \wedge\ c'.size = c.size + 1, \\
&\qquad ite(select(c.c_k, 0) = k, \\
&\qquad\quad i'.pos = 0, \\
&\qquad\quad ite(k \prec \mathrm{select(c.c_k, 0)}, \\
&\qquad\qquad i'.pos = 0 \\
&\qquad\qquad \wedge\ memcpy(c.c_k,\ c'.c_k,\ i'.pos,\ i'.pos + 1,\ c.size - i'.pos) \\
&\qquad\qquad \wedge\ store(c'.c_k,\ i'.pos,\ k) \\
&\qquad\qquad \wedge\ memcpy(c.c_v,\ c'.c_v,\ i'.pos,\ i'.pos + 1,\ c.size - i'.pos) \\
&\qquad\qquad \wedge\ store(c'.c_v,\ i'.pos,\ v) \\
&\qquad\qquad \wedge\ c'.size = c.size + 1, \\
&\qquad\qquad \ldots \\
&\qquad\qquad ite(select(c.c_k, c.size - 1) = k, \\
&\qquad\qquad\quad i'.pos = c.size - 1, \\
&\qquad\qquad\quad ite(k \prec \mathrm{select(c.c_k,\ c.size - 1), i'.pos = c.size - 1, i'.pos = c.size)} \\
&\qquad\qquad\quad \wedge\ memcpy(c.c_k,\ c'.c_k,\ i'.pos,\ i'.pos + 1,\ c.size - i'.pos) \\
&\qquad\qquad\quad \wedge\ store(c'.c_k,\ i'.pos,\ k) \\
&\qquad\qquad\quad \wedge\ memcpy(c.c_v,\ c'.c_v,\ i'.pos,\ i'.pos + 1,\ c.size - i'.pos) \\
&\qquad\qquad\quad \wedge\ store(c'.c_v,\ i'.pos,\ v) \\
&\qquad\qquad\quad \wedge\ c'.size = c.size + 1)\ \ldots\ )))
\end{aligned}
\tag{11}
$$

that induces the following pre- and post-conditions,

$$
\begin{aligned}
&\mathcal{P}((c', i') = c.insert(k, v)) := \\
&\quad k \neq null \\
&\quad \wedge\ v \neq null \\
&\quad \wedge \left(\bigwedge_{j=0}^{c.size-2} select(c.c_k, j) \prec \mathrm{selct(c.c_k, j + 1)} \right)
\end{aligned}
\tag{12}
$$

$$
\begin{aligned}
&\mathcal{Q}((c', i') = c.insert(k, v)) := \\
&\quad i'.i_k = c'.c_k \\
&\quad \wedge\ i'.i_v = c'.c_v \\
&\quad \wedge \left(\bigwedge_{j=0}^{c.size-1} select(c.c_k, j) \neq k \right) \Rightarrow c'.size = c.size + 1 \\
&\quad \wedge \left(\bigwedge_{j=1}^{c.size-1} select(c.c_k, j - 1) \prec \mathrm{selct(c.c_k, j)} \right)
\end{aligned}
\tag{13}
$$

where we assume as pre-conditions $\mathcal{P}$ that v and k must be initialized objects, as well as the order of elements, obey a strict weak ordering rule. Similarly, we assume as post-conditions $\mathcal{Q}$ that the iterator i' will point to the container c', and the strict weak ordering rule will be maintained. We also check whether the size of the container will grow if the key k was not used before; however, this check is bypassed for containers that allow multiple keys.

Remove operations are represented by $c.erase(i)$, where i is an iterator that points to the element to be removed. Similarly to sequential containers (cf. Section 4.2), the model for such operation basically shifts backwards all elements followed by that specific position i. Thus,

$$\begin{aligned}
&\mathcal{C}((c', i') = c.erase(i)) := \\
&\quad memcpy(c.c_k,\ c'.c_k,\ i.pos+1,\ i.pos,\ c.size-(i.pos+1)) \\
&\quad \wedge\ memcpy(c.c_v,\ c'.c_v,\ i.pos+1,\ i.pos,\ c.size-(i.pos+1)) \\
&\quad \wedge\ c'.size = c.size-1 \\
&\quad \wedge\ i'.pos = i.pos+1
\end{aligned} \tag{14}$$

that induces the following pre- and post-conditions,

$$\begin{aligned}
&\mathcal{P}((c', i') = c.erase(i)) := \\
&\quad i.i_k \neq null \\
&\quad i.i_v \neq null \\
&\quad \wedge\ 0 \leq i.pos < c.size \\
&\quad \wedge\ c.size \neq 0 \Rightarrow (c.c_k \neq null \wedge c.c_v \neq null)
\end{aligned} \tag{15}$$

$$\begin{aligned}
&\mathcal{Q}((c', i') = c.erase(i)) := \\
&\quad i'.i_k = c'_k \\
&\quad \wedge\ i'.i_v = c'_v \\
&\quad \wedge\ select(c'.c_k,\ i'.pos) = select(c.c_k,\ i.pos+1) \\
&\quad \wedge\ select(c'.c_v,\ i'.pos) = select(c.c_v,\ i.pos+1)
\end{aligned} \tag{16}$$

which have similar properties as the ones held by the *erase* method from sequential containers, except that $i'.i_k$ must point to the position immediately after the erased one and the equivalency of $c'.c_k$ and $i'.i_k$. Finally, search operations over associative containers are modelled by a container c with a method call $c.search(k)$. Then, if an element with key k is found, the method returns an iterator that points to the corresponding element; otherwise, it returns an iterator that points to the position immediately after the last container's element. Hence,

$$\begin{aligned}
&\mathcal{C}((c', i') = c.search(k)) := \\
&\quad ite(c.size = 0, \\
&\qquad i'.pos = c.size, \\
&\qquad ite(select(c.c_k, 0) = k, \\
&\qquad\quad i'.pos = 0, \\
&\qquad\quad ... \\
&\qquad\quad ite(select(c.c_k, c.size-1) = k, \\
&\qquad\qquad i'.pos = c.size-1, \\
&\qquad\qquad i'.pos = c.size) \ldots))
\end{aligned} \tag{17}$$

that induces the following pre- and post-conditions,

$$\begin{aligned}
&\mathcal{P}((c', i') = c.search(k)) := \\
&\quad k \neq null
\end{aligned} \tag{18}$$

$$
\begin{aligned}
&\mathcal{Q}((c', i') = c.search(v)) := \\
&\quad c'.c_k = c.c_k \\
&\quad \wedge\ c'.c_v = c.c_v \\
&\quad \wedge\ c'.size = c.size \\
&\quad \wedge\ i'.i_k \neq c'.c_k \\
&\quad \wedge\ i'.i_v \neq c'.c_v \\
&\quad \wedge\ ite(select(i'.i_k,\ i'.pos)\ =\ select(c'.c_k, i'.pos), \\
&\qquad\quad select(i'.i_k,\ i'.pos) = k, \\
&\qquad\quad select(i'.i_k,\ i'.pos) = select(c'.c_k,\ c'.size)) \\
&\quad \wedge\ ite(select(i'.i_v,\ i'.pos)\ =\ select(c'.c_v, i'.pos), \\
&\qquad\quad select(i'.i_v,\ i'.pos) = v, \\
&\qquad\quad select(i'.i_v,\ i'.pos) = select(c'.c_v,\ c'.size))
\end{aligned}
\tag{19}
$$

that are also similar to the properties held by the *search* operation from sequential containers, except that the search happens over keys.

5 | EXCEPTION HANDLING

Exceptions are unexpected circumstances that arise during the execution of a program, e.g., runtime errors [47]. In C++, the exception handling is split into three (basic) elements: a **`try`** block, where a thrown exception can be directed to a **`catch`** statement; a set of **`catch`**statements, where a thrown exception can be handled; and a **`throw`** statement that raises an exception.

To accurately define the verification of exception handling in C++, we formally define two syntactic domains, including exceptions E and handlers H as follows:

$$
\begin{aligned}
E &:= e \mid e_{[]} \mid e_{f()} \mid e_{*} \mid e_{null} \\
H &:= h \mid h_{[]} \mid h_{f()} \mid h_{*} \mid h_{v} \mid h_{...} \mid h_{null}
\end{aligned}
$$

In this context, e and h are classes of variables of type E and H, respectively. We use the notation $e_{[]}$ to denote a thrown exception of type array, $e_{f()}$ is a thrown exception of type function, e_{*} is a thrown exception of type pointer, and e_{null} is an empty exception used to track when a *throw* expression does not throw anything. Similarly, we use the notation $h_{[]}$ to denote a **`catch`** statement of type array, $h_{f()}$ is a **`catch`** statement of type function, h_{*} is a **`catch`** statement of type pointer, h_{v} is a **`catch`**statement of type void pointer (i.e., **`void *`**), $h_{...}$ is a **`catch`** statement of type ellipsis [45], and h_{null} is an invalid **`catch`** statement used to track when a thrown exception does not have a valid handler.

Based on such domains, we must define a 2-arity predicate $M(e, h)$, which evaluates whether the type of thrown exception e is compatible with the type of a given handler h as shown in Equation (20). Furthermore, we declare the unary function $\zeta : H^{*} \mapsto H$ that removes qualifiers **`const`**, **`volatile`**, and **`restrict`** from the type of a **`catch`** statement c. We also define the 2-arity predicates unambiguous base $U(e, h)$ and implicit conversion $Q(e, h)$. On one hand, $U(e, h)$ determines whether the type of a **`catch`** statement h is an unambiguous base [45] for the type of a thrown exception e as shown in Equation (21). On the other hand, $Q(e, h)$ determines whether a thrown exception e can be converted to the type of the **`catch`** statement h, either by qualification or standard pointer conversion [45] as shown in Equation (22).

$$
M(e,h) \stackrel{\text{def}}{=} \begin{cases} \top, & \textit{type of } e \textit{ is matches to the type of } h \\ \bot, & \textit{otherwise} \end{cases} \tag{20}
$$

$$
U(e,h) \stackrel{\text{def}}{=} \begin{cases} \top, & c \textit{ is an unambiguous base of } e \\ \bot, & \textit{otherwise} \end{cases} \tag{21}
$$

$$
Q(e,h) \stackrel{\text{def}}{=} \begin{cases} \top, & e \textit{ can be implicit converted to } h \\ \bot, & \textit{otherwise} \end{cases} \tag{22}
$$

The C++ language standard defines rules to connect **throw** expressions and **catch**statements [45], which are all described in Table 2. Each rule represents a function $r_k : E \mapsto H$ for $k = [1 \,..\, 9]$, where a thrown exception e is mapped to a valid **catch** statement h. ESBMC evaluates every thrown exception e against all rules and all **catch** statements in the program through the $(n+1)$-arity function handler $\mathcal{H}$. As shown in Equation (23), after the evaluation of all rules (i.e., $h_{r_1}, \ldots, h_{r_9}$), ESBMC returns the first handler h_{r_k} that matched the thrown exception e.

$$\begin{aligned}\mathcal{H}(e, h_1, \ldots, h_n) := \\ & h_{r_1} = r_1(e, h_1, \ldots, h_n) \\ & \wedge \ldots \\ & \wedge\ h_{r_9} = r_9(e, h_1, \ldots, h_n) \\ & \wedge\ ite(h_{r_1} \neq h_{null}, h_{r_1}, \\ & \quad ite(h_{r_2} \neq h_{null}, h_{r_2}, \\ & \quad \ldots \\ & \quad ite(h_{r_9} \neq h_{null}, h_{r_9}, h_{null}) \ldots)\end{aligned} \tag{23}$$

To support exception handling in ESBMC, we extended our GOTO conversion code and the symbolic engine. In the former, we had to define new instructions and model the throw expression as jumps. In the latter, we implemented the rules for throwing and catching exceptions, as shown in Table 2, and the control flows for the unexpected and terminate handlers (cf. Section 5.2).

The GOTO conversion slightly modifies the exception handling blocks H. The following instructions model a **try** block: a **CATCH** instruction to represent the start of the **try** block, the instructions representing the code inside the **try** block, a **CATCH** instruction to represent the end of the **try** block and a GOTO instruction targeting the instructions after the **try** block. Each catch statement is represented using a label, the instructions representing the exception handling and a GOTO instruction targeting the instructions after the **catch** block.

We use the same **CATCH** instruction to mark the beginning and end of the **try** block. However, **CATCH** instructions at the beginning and at the end differ by the information they hold; the **CATCH** instruction that marks the beginning of a **try** block has a map from the types of the catch statements and their labels in the GOTO program, while the second **CATCH** instruction has an empty map. The GOTO instruction targeting the instructions after the **catch** block shall be

TABLE 2 Rules to connect **throw** expressions and **catch** blocks

Rule	Behaviour	Formalization
r_1	Catches an exception if the type of the thrown exception e is equal to the type of the **catch**h.	$ite(\exists h \cdot\ M(e,h), h_{r_1} = h, h_{r_1} = h_{null})$
r_2	Catches an exception if the type of the thrown exception e is equal to the type of the **catch**h, ignoring the qualifiers *const*, *volatile*, and *restrict*.	$ite(\exists h \cdot\ M(e, \zeta(h)), h_{r_2} = h, h_{r_2} = h_{null})$
r_3	Catches an exception if its type is a pointer of a given type x and the type of the thrown exception is an array of the same type x.	$ite(\exists h \cdot\ e = e_{[]} \wedge h = h_* \wedge M(e_{[]}, h_*), h_{r_3} = h_*, h_{r_3} = h_{null})$
r_4	Catches an exception if its type is a pointer to function that returns a given type x and the type of the thrown exception is a function that returns the same type x.	$ite(\exists h \cdot\ e = e_{f()} \wedge h = h_{f()} \wedge M(e_{f()}, h_{f()}), h_{r_4} = h_{f()}, h_{r_4} = h_{null})$
r_5	Catches an exception if its type is an unambiguous base type for the type of the thrown exception.	$ite(\exists h \cdot\ U(e,h), h_{r_5} = h, h_{r_5} = h_{null})$
r_6	Catches an exception if the type of the thrown exception e can be converted to the type of the **catch**h, either by qualification or standard pointer conversion [45].	$ite(\exists h \cdot\ e = e_* \wedge h = h_* \wedge Q(e_*, h_*), h_{r_6} = h_*, h_{r_6} = h_{null})$
r_7	Catches an exception if its type is a void pointer h_v and the type of the thrown exception e is a pointer of any given type.	$ite(\exists h \cdot\ e = e_* \wedge h = h_v, h_{r_7} = h_v, h_{r_7} = h_{null})$
r_8	Catches any thrown exception if its type is ellipsis.	$ite(\forall e \cdot \exists h \cdot\ h = h_{\ldots}, h_{r_8} = h_{\ldots}, h_{r_8} = h_{null})$
r_9	If the **throw** expression does not throw anything, it should re-throw the last thrown exception e_{-1}, if it exists.	$ite(e = e_{null} \wedge e_{-1} \neq e_{null},$ $h'_{r_1} = r_1(e_{-1}, h_1, \ldots, h_n)$ $\wedge\ \ldots$ $\wedge\ h'_{r_9} = r_9(e_{-1}, h_1, \ldots, h_n),$ $h_{r_9} = h_{null})$

```
1  int main ()
2  {
3    try {
4      if (nondet())
5        throw 20;
6      else
7        throw 10.0f;
8    }
9    catch (int i) {
10     assert(i == 20);
11   } catch (float f) {
12     assert(f == 10.0);
13   }
14
15   return 0;
16 }
```

try block

catch block

(a) Try-catch example of throwing an integer exception.

```
1  main() (c::main):
2     CATCH signed_int->3, float->4
3     FUNCTION_CALL:
4        return_value_nondet$1=nondet()
5     IF !return_value_nondet$1 THEN GOTO 1
6     THROW signed_int: 20
7     GOTO 2
8  1: THROW float: 10f
9  2: CATCH
10    GOTO 5
11 3: signed int i;
12    ASSERT i == 20
13    GOTO 5
14 4: float f;
15    ASSERT f == 10f
16 5: RETURN: 0
17 END_FUNCTION
```

(b) GOTO instructions.

FIGURE 10 Example of try-catch conversion to GOTO instructions

called in case no exception is thrown. The GOTO instructions at the end of each **`catch`** are called so that only the instructions of the current **`catch`** is executed, as shown in Figure 10.

During the SSA generation, when the first **`CATCH`** instruction is found, the map is stacked because there might be nested **`try`** blocks. If an exception is thrown, ESBMC encodes the jump to a catch statement according to the rules defined in Table 2, including a jump to an invalid **`catch`** that triggers a verification error; that is, it represents an exception thrown that cannot be caught. If a suitable exception handler is found, then the thrown value is assigned to the **`catch`** variable (if any); otherwise, if there exists no valid exception, an error is reported. If the second **`CATCH`** instruction is reached and no exception was thrown, the map is freed for memory efficiency. The **`try`** block is handled as any other block in a C++ program. Destructors of variables in the stack are called by the end of the scope. Furthermore, by encoding throws as jumps, we also correctly encode memory leaks. For example, suppose an object is allocated inside a **`try`** block, and an exception is thrown and handled. In that case, it will leak unless the reference to the allocated memory is somehow tracked and freed.

Our symbolic engine also keeps track of *function frames*, that is, several pieces of information about the function it is currently evaluating, including arguments, recursion depth, local variables, and others. These pieces of information are essential not only because we want to handle recursion or find memory leaks but also allow us to connect exceptions thrown outside the scope of a function and handle exception specification (as described in Section 5.1).

5.1 | Exception specification

The exception specification (illustrated in Figure 11) defines which exceptions can be thrown by a function or method (including constructors). It is formed by an exception list and can be empty, i.e., the function or method cannot throw an exception. Exceptions thrown and handled inside a function or method are not affected by the exception specification.

```
1  /* function 1 can throw exceptions
2     of type int and float */
3  void func1() throw(int, float) {
4    ...
5  }
6  /* function 2 can't throw an exception */
7  void func2() throw() {
8    try {
9      /* OK, exception handled inside func2's scope */
10     throw 1;
11   }
12   catch(int) {
13     /* error handling for integer exceptions */
14   }
15 }
```

FIGURE 11 Example of exception specification

To support the verification of programs with exception specifications, an instruction **THROW_DECL** is inserted at the beginning of the given function or method. This instruction contains a list of allowed exceptions that are checked whenever an exception is thrown outside the scope of the function or method. Similar to the **catch** map, they are stacked due to the possibility of nested exception specifications and are freed at the end of the function or method.

An exception thrown from inside a function follows the same rules defined in Table 2. Exception specifications check any exception thrown outside the function scope. If the type of the exception was not declared in the exception specialization, a different exception is raised and a separate path in the program is taken: the unexpected handler.

5.2 | Terminate and unexpected handlers

During the exception handling process, errors can occur, causing the process to be aborted for any given reason (e.g., throwing an exception outside a **try** block or not catching a thrown exception). When this happens, the terminate handler is called.

Figure 12a shows the terminate handler implementation. The terminate handler is a function that has the default behaviour of calling the **abort** function. However, this behaviour can be slightly changed by the developer, using the function **set_terminate(f)**, where *f* is a function pointer to a function that has no parameter and no return value (type **void**). By setting the new terminate function, it will be called before the **abort** function.

For the verification of programs that override the terminate handler, we define a function **__default_terminate()**, as illustrated in Figure 12a, that contains the default termination behaviour, calling **abort**. ESBMC also keeps a global function pointer to the terminate function, which can either point to the default behaviour or the user-defined behaviour. Finally, when the terminate function is called, we should guarantee that the **abort** function will be called, even if the terminate function is replaced (as shown in label *E* in Figure 12a).

However, there is one case where the unexpected handler is called instead of the terminate handler. When an exception not allowed by the exception specification (Section 5.1) is thrown by a function or method, when this happens, the unexpected handler is called.

```
1  namespace std {
2    // A) function definition:
3    //      no return, no parameters
4    typedef void (*terminate_handler)();
5
6    // B) Default terminate function:
7    //      calls abort
8    void __default_terminate() throw () {
9      // Aborts the program with
10     // the message "Aborted"
11     __ESBMC_assert(0, "Aborted");
12   }
13
14   // C) Set the default behavior
15   terminate_handler terminate_pf =
16     __default_terminate;
17
18   // D) Set the user defined
19   //      terminate function
20   terminate_handler
21   set_terminate(terminate_handler f)
22   throw () {
23     terminate_pf=f;
24   }
25
26   // E) Model for terminate function:
27   //      calls current
28   // terminate handler function
29   void terminate() {
30     terminate_pf();
31     __default_terminate();
32   }
33 }
```

(a) Terminate functions: A) function type definition; B) Default terminate behavior; C) Set the default behavior; D) Function *set_terminate*; E) Model for function terminate.

```
1  namespace std {
2    // A) function definition:
3    //      no return, no parameters
4    typedef void (*unexpected_handler)();
5
6    // Default unexpected function:
7    // calls terminate
8    void __default_unexpected() throw () {
9      __default_terminate();
10   }
11
12   // C) Set the default behavior
13   unexpected_handler unexpected_pf =
14     default_unexpected;
15
16   // D) Set the user defined
17   //      unexpected function
18   unexpected_handler
19   set_unexpected(unexpected_handler f)
20   throw () {
21     unexpected_pf = f;
22   }
23
24   // E) Model for unexpected function:
25   // calls unexpected handler function
26   void unexpected() {
27     unexpected_pf();
28     throw;
29   }
30 }
```

(b) Unexpected functions: A) function type definition; B) Default unexpected behavior; C) Set the default behavior; D) Function *set_unexpected*; E) Model for function unexpected.

FIGURE 12 Examples of terminate and unexpected handlers

```
1  #include <exception>
2  #include <cassert>
3  using namespace std;
4
5  void myunexpected () {
6    throw;
7  }
8
9  void myfunction () throw (int, bad_exception) {
10   throw 'x';
11 }
12
13 int main (void) {
14   set_unexpected (myunexpected);
15   try {
16     myfunction();
17   }
18   catch (int) { assert(0); }
19   catch (bad_exception be) { return 1; }
20   return 0;
21 }
```

FIGURE 13 Fragment of code using bad exception

The unexpected handler works similarly to the terminate handler. It will either call terminate or re-throw the not allowed exception. Similar to **`set_terminate`**, there exists a function **`set_unexpected(f)`**, where **`f`** is function pointer to a function that has no parameter and no return value (type **`void`**).

Figure 12b illustrates the unexpected handler implementation. The default behaviour is to re-throw the thrown exception, and, as the exception specification already forbids it, we should call terminate to finish the program. ESBMC also keeps a global function pointer to the unexpected function, which either points to the default behaviour or the user-defined behaviour. If the unexpected handler was replaced, we must still guarantee that an exception will be thrown, so the forbidden exception will be re-thrown (as shown in line 27 in Figure 12b). If the replaced unexpected function throws an exception that is not forbidden by the function, the code will not terminate.

Finally, we also need to model the unexpected behaviour when using **`bad_exception`**. Figure 13 shows an example of code using **`bad_exception`**. In this example, the user replaced the unexpected function with a function containing a re-throw. The code then calls **`myfunction()`**, which tries to throw a forbidden **`char`** exception. At this moment, **`myunexpected`** function is called and tries to re-throw the **`char`** exception, which is forbidden. ESBMC matches the compiler's behaviour and checks whether **`bad_exception`** is one of the allowed exceptions in the exception specification; if this is true, a **`bad_exception`** exception will be thrown instead of the original forbidden exception.

6 | EXPERIMENTAL EVALUATION

Our experimental evaluation compares ESBMC against LLBMC and DIVINE regarding correctness and performance in the verification process of C++03 programs; DIVINE was developed by Baranová et al. [16], and LLBMC was developed by Merz, Falke, and Sinz [12]. Section 6.1 shows a detailed description of all tools, scripts, and benchmark dataset, while Section 6.2 presents the results and our evaluation. Our experiments are based on a set of publicly available benchmarks. All tools, scripts, benchmarks, and results of our evaluation are available on a replication package [58], including all data to generate the percentages. More information about ESBMC is also available at the project's webpage https://esbmc.org/.

6.1 | Experimental design, materials and methods

Our experiments aim at answering two experimental questions regarding *correctness* and *performance* of ESBMC:

i. (EQ-I) How accurate is ESBMC when verifying the chosen C++03 programs?
ii. (EQ-II) How does ESBMC performance compare to other existing model checkers?

To answer both questions, we evaluate all benchmarks with ESBMC *v*2.1, DIVINE *v*4.3, and LLBMC *v*2013.1. ESBMC *v*2.1 contains the last stable version of our C++ front-end, since the changes necessary to introduce a new C

front-end on ESBMC *v*3.0 were disruptive. The new C front-end is based on the clang's AST [22], which completely changes the way ESBMC processes source files. Update the C++ front-end to also use clang's AST is part of our future work (cf. Section 8). We use LLBMC *v*2013.1 in our evaluation since it is the latest publicly available version of the LLBMC tool. We also applied CBMC [25] (*v*5.3) in our benchmark set. However, we do not detail the results in the experimental evaluation because the tool aborts during parser in 1500 cases and reproduces false-negative results in the remaining 3. The vast majority of our benchmarks use STL functionalities, which CBMC does not support. The lack of support for C++ features in CBMC was also reported by Merz et al. [12], Monteiro et al. [18], and Ramalho et al. [24].

To tackle modern aspects of the C++ language, the comparison is based on a benchmark dataset that consists of 1513 C++03 programs. In particular, 290 programs were extracted from the book "C++ How to Program" [47], 432 were extracted from C++ Resources Network [59], 16 were extracted from NEC Corporation [60], 16 programs were obtained from LLBMC [12], 39 programs were obtained from CBMC [25], 55 programs were obtained from the GCC test suite [42], and the others were developed to check several features of the C++ programming language [24]. The benchmarks are split into 18 test suites: *algorithm* contains 144 benchmarks to check the Algorithm library functionalities; *cpp* contains 357 general benchmarks, which involves C++03 libraries for general use, such as I/O streams and templates; this category also contains the LLBMC benchmarks and most NEC benchmarks. The test suites *deque* (43), *list* (72), *queue* (14), *stack* (14), *priority_queue* (15), *stream* (66), *string* (233), *vector* (146), *map* (47), *multimap* (45), *set* (48), and *multiset* (43) contain benchmarks related to the standard template containers. The category *try_catch* contains 81 benchmarks to the exception handling and the category *inheritance*contains 51 benchmarks to check inheritance and polymorphism mechanisms. Finally, the test suites *cbmc* (39), *templates* (23), and *gcc-template* (32) contain benchmarks from the GCC[4] and CBMC[5] test suite, which are specific to templates.

Each benchmark is tested and manually inspected in order to identify and label bugs. Thus, 543 out of the 1513 benchmarks contain bugs (i.e., 35.89%) and 970 are bug-free (i.e., 64.11%). This inspection is essential to compare verification results from each model checker and properly evaluates whether real errors were found. We evaluate three types of properties: (i) memory-safety violations (e.g., arithmetic overflow, null-pointer dereferences, and array out-of-bounds), (ii) user-specified assertions, and (iii) proper use of C++ features (e.g., exception-handle violations). We only exclude LLBMC from the evaluation of exception handling since the tool does not support this feature. All tools support all the remaining features and properties under evaluation.

All experiments were conducted on a computer with an i7-4790 processor, 3.60GHz clock, with 16GB RAM and Ubuntu 14.04 64-bit OS. ESBMC, LLBMC, and DIVINE were set to a time limit of 900 s (i.e., 15 min) and a memory limit of 14GB. All presented execution times are CPU times; that is, only the elapsed periods spent in the allocated CPUs. Furthermore, memory consumption is the amount of memory that belongs to the verification process and is currently present in RAM (i.e., not swapped or otherwise not-resident). Both CPU time and memory consumption were measured with the `times` system call (POSIX system). Neither swapping nor turbo boost was enabled during experiments and all executed tools were restricted to a single process.

The tools were executed using three scripts: the first one for ESBMC,[6] which reads its parameters from a file and executes the tool; the second one for LLBMC, which first compiles the program to bitcode, using clang,[7] then it reads the parameters from a file and executes the tool[8]; and the last one for DIVINE, which also first pre-compiles the C++ program to bitcode, then performs the verification on it.[9] The loop unrolling defined for ESBMC and LLBMC (i.e., the *B* value) depends on each benchmark. In order to achieve a fair comparison with ESBMC, an option from LLBMC had to be disabled. LLBMC does not support exception handling and all bitcodes were generated without exceptions (i.e., with the `-fno-exceptions` flag of the compiler). If exception handling is enabled, then LLBMC always aborts the verification process.

6.2 | Results and discussion

In this section, we present the results using percentages (concerning the 1513 C++ benchmarks), as shown in Figure 14. *Correct* represents the positive results, that is, percentage of benchmarks with and without bugs correctly verified. *False positives* represent the percentage of benchmarks reported as correct, but they are incorrect; similarly, *False negatives* represent the percentage of benchmarks reported as incorrect, but that are correct. Finally, *Unknown* represents the

[4]https://github.com/nds32/gcc/tree/master/gcc/testsuite/
[5]https://github.com/diffblue/cbmc/tree/develop/regression
[6]esbmc *.cpp –unwind *B* –no-unwinding-assertions -I /libraries/
[7]clang++ -c -g -emit-llvm *.cpp -fno-exceptions -o main.bc
[8]llbmc *.o -o main.bc –ignore-missing-function-bodies –max-loop-iterations=*B* –no-max-loop-iterations-checks
[9]divine verify *.cpp

benchmarks where each tool aborted the verification process due to internal errors, timeout (i.e., the tool was killed after 900 s) or a memory out (i.e., exhausted the maximum memory allowed of 14GB). In the Exception Handling category, LLBMC is excluded since it does not support this feature; if exception handling is enabled, then LLBMC continuously aborts the verification process. Furthermore, to better present the results of our experimental evaluation, the test suites were grouped into four categories:

- Standard Containers—formed by *algorithm*, *deque*, *vector*, *list*, *queue*, *priority_queue*, *stack*, *map*, *multimap*, *set* and *multiset* test suites (631 benchmarks);
- Inheritance and Polymorphism—formed by the *inheritance* test suite (51 benchmarks).
- Exception Handling—formed by the *try_catch* test suite (81 benchmarks);
- C++03—formed by *cpp*, *string*, *stream*, *cbmc*, *gcc-templates* and *templates* test suites (750 benchmarks).

On the Standard Containers category (see Figure 14), ESBMC presented the best results and reached a successful verification rate of 78.45%, while LLBMC reported 70.36% and DIVINE 44.69%. ESBMC's noticeable results for containers are directly related to its COM. The majority of the benchmarks for this category contain standard assertions to map the support of container-based operations, for example, to check whether the `operator[]` from a `vector` object is called with an argument out of range, which is undefined behaviour [45]. We place standard C++ assertions in the benchmarks to evaluate how each verifier handles container-based operations. ESBMC reports a false-positive rate of 2.54% and a false-negative rate of 8.87%, which is due to internal implementation issues during pointer encoding (cf. Section 4). We are currently working to address them in future versions. ESBMC also reported 10.14% of unknown results due to limitations in templates-related features such as SFINAE [45] and nested templates. LLBMC reports a false-positive rate of 2.85% and a false-negative rate of 17.60%, mostly related to erroneously evaluating assertions (e.g., assertions to check whether a container is empty or it has a particular size). It also reports an unknown rate of 9.19% regarding timeouts, memory outs, and crashes when performing formula transformation [12]. DIVINE does not report any timeout, memory out, or false-positive results for this category, but an expressive false-negative rate of 49.92%, resulting from errors to check assertions (similarly to LLBMC). DIVINE also reports an unknown rate of 5.39% due to errors with pointer handling, probably due to imprecise (internal) encoding.

On the Inheritance and Polymorphism category (see Figure 14), ESBMC presented the best results and reached a successful verification rate of 84.32% while LLBMC reported 68.63% and DIVINE 54.90%. ESBMC does not report any timeout or memory out, but it reports a false-negative rate of 15.68%, due to implementation issues to handle pointer encoding. LLBMC does not report any false positives, timeouts, or memory outs results. However, it reports a false-negative rate of 5.88%, which is related to failed assertions representing functional aspects of inherited classes. It also reported an unknown rate of 25.49% regarding multiple inheritance. DIVINE does not report any timeout, memory out, or false-positive results for this category, but a false-negative rate of 23.53% and an unknown rate of 21.57%, which is a result of errors when handling dynamic casting, virtual inheritance, multiple inheritance, and even basic cases of inheritance and polymorphism.

On the Exception Handling category (see Figure 14), ESBMC presented the best results and reached a successful verification rate of 87.66% while DIVINE reported 62.96%. ESBMC does not report any timeout or memory out, but it reports a false-positive rate of 3.70% and a false-negative rate of 2.47%. These bugs are related to the implementation of rule r_6 from Table 2 in ESBMC, that is, "catches an exception if the type of the thrown exception e can be converted to the type of the catch h, either by qualification or standard pointer conversion"; we are currently working on fixing these issues. ESBMC also presents an unknown rate of 3.70% due to previously mentioned template limitations. DIVINE does not report any timeout or memory out. However, it reports a false-positive rate of 7.40% and a false-negative rate of 17.30%. It incorrectly handles re-throws, exception specification, and the unexpected as well as terminate function handlers. DIVINE also presents an unknown rate of 12.34% due to errors when dealing with exceptions thrown by derived classes, instantiated as base classes, which is probably related to the imprecise encoding of *vtables*.

To evaluate how these model checkers perform when applied to general C++03 benchmarks, we evaluate them against the category C++03. In this category, model checkers deal with benchmarks that make use of the features discussed in this paper (e.g., exception handling and containers), and a wider range of libraries from the STL, manipulation of strings and streams, among other C++03 features. ESBMC presented the highest successful verification rate, 89.20%, followed by DIVINE 67.20% and LLBMC 62.27%. The successful expressive rate of ESBMC in this category not only correlates to its support for core C++03 features (i.e., templates, inheritance, polymorphism, and exception handling) or its ability to check functional aspects of the standard containers but also because COM contains abstractions for all standard libraries shown in Table 1. For instance, the operational model for the string library enables ESBMC to achieve a success rate of 99.14% in the *string* test suite, which contains benchmarks that target all methods provided in C++03 for `string` objects. Note that running ESBMC without COM over the benchmarks, 98.08% fail since the majority uses at least one standard template library. ESBMC does not report any memory out, but it reports a

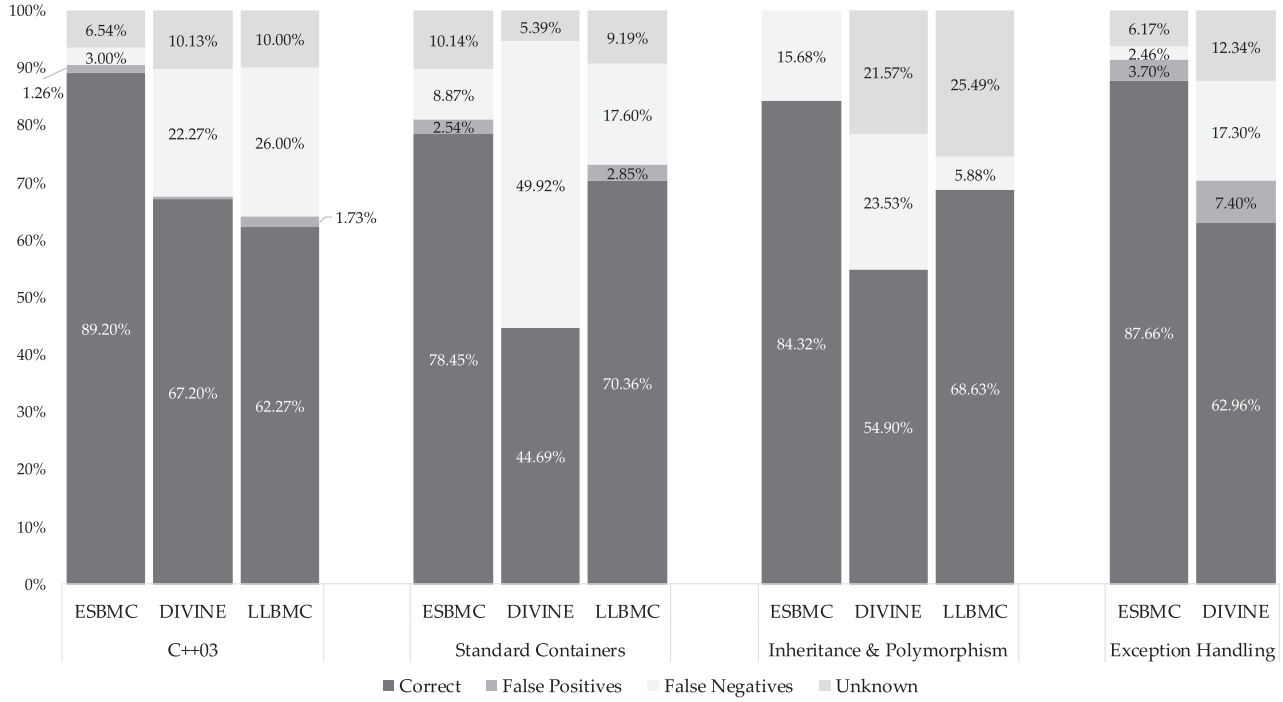


FIGURE 14 Experimental results for ESBMC *v*2.1, DIVINE *v*4.3, and LLBMC *v*2013.1

false-positive rate of 1.26%, a false-negative rate of 3.00%, and an unknown rate of 6.54%, which are all due to the same issues pointed by the previous experiments. DIVINE does not report any false positives, timeout, or a memory out, but a false-negative rate of 22.27%, which is a result of errors when checking assertions representing functional properties of objects across all STL (similar to LLBMC). DIVINE reports one false positive regarding the instantiation of function template specialization and an unknown rate of 10.13% due to crashes when handling pointers. LLBMC reports a false-positive rate of 1.73% and a false-negative rate of 26.00%, which is related to errors when checking assertions that represent functional properties of objects (e.g., asserting the size of a **`string`** object after an operation) or dealing with **`stream`** objects in general. It also reported an unknown rate of 10.00%, mainly regarding operator overloading errors and the ones mentioned in the previous categories.

A small number of counterexamples generated by the three tools were manually checked, but we understand that this is far from ideal. The best approach is to use an automated method to validate the counterexample, such as the witness format proposed by Beyer et al. [61]; however, the available witness checkers do not support the validation of C++ programs. Implementing such a witness checker for C++ would represent a significant development effort, which we leave it for future work.

Figure 15 illustrates the accumulative verification time and memory consumption for the tools under evaluation. All the tools take more time to verify the test suites *algorithm*, *string*, and *cpp*, due to a large number of test cases and the presence of pointers and iterators. ESBMC is the fastest of the three tools, 3.2 times faster than LLBMC and only 155.7 s faster than DIVINE. In terms of verification time, DIVINE is the only tool that did not use more than the defined limit of 900 s, while ESBMC and LLBMC aborted due to timeout in 4 and 25 benchmarks, respectively. DIVINE is the only tool that did not use more than the defined limit of 14GB per benchmark in terms of memory consumption. At the same time, ESBMC and LLBMC aborted due to exhaustion of the memory resources in 3 and 11 of them, respectively. Even so, LLBMC consumes less memory overall (614.92GB) when compared with DIVINE (627.97GB) and ESBMC (2210.91GB).

Overall, ESBMC achieved the highest success rate of 84.27% in 15,761.90 s (approximately 4 h and 23 min), faster than LLBMC and DIVINE, which positively answers our experimental questions EQ-I and EQ-II. LLBMC correctly verified 62.52% in 50,564.10 s (approximately 14 h) and can only verify the programs if exception handling is disabled, which is not a problem for both ESBMC and DIVINE. DIVINE correctly verified 57.17% in 15,917.60 s (approximately 4 h and 26 min). Regarding memory usage, ESBMC has the highest usage among the three tools, which is approximately 3.5 times higher than DIVINE and LLBMC, respectively. This high consumption is due to the generation process of SSA forms (cf. Section 3). However, its optimization is under development for future versions.

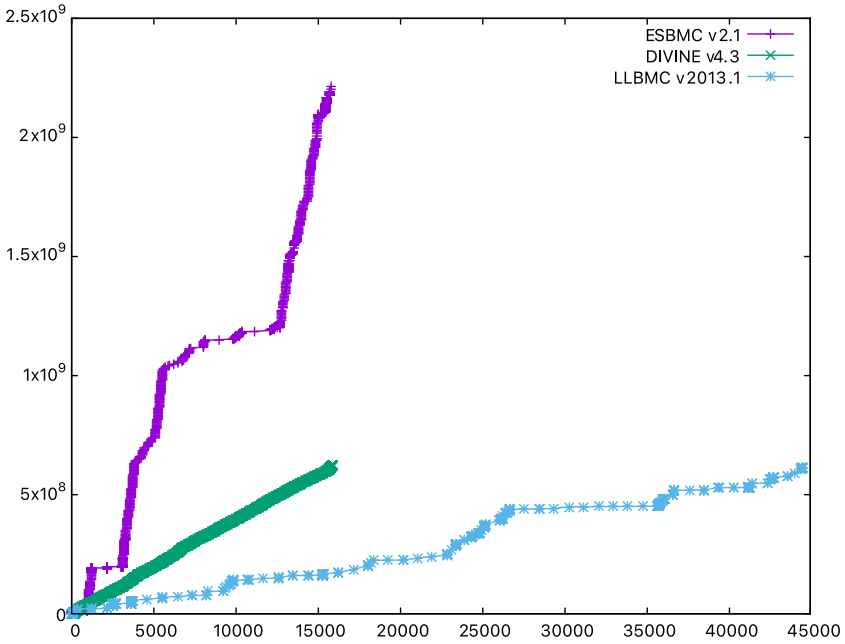


FIGURE 15 Comparison of accumulative verification time and accumulative memory consumption among ESBMC *v*2.1, DIVINE *v*4.3, and LLBMC *v*2013.1 throughout the verification process of all benchmarks

In conclusion, our experimental evaluation indicates that ESBMC outperforms two state-of-the-art model checkers, DIVINE and LLBMC, regarding the verification of inheritance, polymorphism, exception handling, and standard containers. The support for templates in ESBMC needs improvements. However, the current work-in-progress clang front-end will not only cover this gap (because clang will instantiate all the templates in the program) but will also allow ESBMC to handle new versions of the language (e.g., C++11). Even with its current support for templates, our experimental results allow us to conclude that ESBMC represents the state-of-the-art regarding applying model checking in C ++ programs.

6.3 | Sniffer application

This section describes the results of the verification process using ESBMC and LLBMC in a *sniffer* program. We were unable to use DIVINE to verify the code because the tool does not offer support for the verification of some libraries used in the program (e.g., **`boost`**[10]), which makes the verification process an infeasible task; that is, DIVINE would report incorrect results. Nokia Institute of Technology (INdT) made this program available. Sniffer is responsible for capturing and monitoring traffic conditions of a network, which supports Message Transfer Part Level 3 User Adaptation Layer (M3UA) [62]. This service offers the transport of SS7 protocols (Signalling System *No*.7) and makes use of the services provided by the Stream Control Transmission Protocol (SCTP). The *Sniffer*program contains 20 classes, 85 methods, and 2800 lines of C++ code.

The following properties were verified in the *sniffer* program: arithmetic underflow and overflow, division by zero, and array bounds violation. Due to confidentiality issues, we were only able to verify 50 of 85 methods since INdT did not provide some classes required by the unverified methods. From the verified code, ESBMC was able to identify five errors, related to arithmetic under- and overflow while LLBMC was able to identify only three of them. All errors were reported to developers, who confirmed them. As an example of an error found, Figure 16 shows the **`getPayloadSize`** method from the **`PacketM3UA`** class. In this method, an arithmetic overflow can occur. The method returns **`ntohs`**, an **`unsigned int`**, but the **`getPayloadSize`** method must return a **`signed int`**. In this case, a possible solution is to change the return type of the **`getPayloadSize`** method to **`unsigned int`**.

[10]https://www.boost.org/

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

FIGURE 16 Arithmetic overflow in the `typecast` operation from the `getPayloadSize` method

TABLE 3 Related work comparison

		C++ programming language support			
Related work	**Conversion to intermediate languages**	**Templates**	**Standard Template Libraries**	**Inheritance and Polymorphism**	**Exception Handling**
LLBMC [65]	LLVM	Yes	Yes	Yes	No
MCP [65]	LLVM	Yes	Yes	Yes	YES
SATABS [14]	No	Yes	Yes	No	No
CBMC [25]	No	Yes	No	No	No
DIVINE [16]	LLVM	Yes	Yes	Yes	Yes
ESBMC v2.0 [24]	**No**	**Yes**	**Yes**	**Yes**	**Yes**

7 | RELATED WORK

Conversion of C++ programs into another language makes the verification process easier since C++ model checkers are still in the early development stages. There are more stable verification tools written for other programming languages, such as C [10]. Numerous verification tools use the LLVM infrastructure to verify programs (e.g., SMACK [17], DIVINE [16], and Seahorn [63]), often verifying the LLVM bitcode or converting it to an intermediate representation (e.g., Boogie [64]). Such approaches pose a few challenges. For instance, undefined constructs (according to the C/C++ specification) are baked into the bitcode (e.g., the order in which function call arguments are evaluated). In addition, types are also baked in the bitcode. Note that this is less of an issue since we have to fix it on a bit-width implementation. However, context information might be lost (e.g., variable, class, and function names are mangled in C++) or source location information.

When it comes to the verification of C++ programs, most of the model checkers available in the literature focus their verification approach on specific C++ features, such as exception handling, and end up neglecting other features of equal importance, such as the verification of the STL [66,67]. Table 3 shows a comparison among other studies available in the literature and our approach.

Merz, Falke, and Sinz [12,65] describe LLBMC, a tool that uses BMC to verify C++ programs. The tool first converts the program into LLVM intermediate representation, using clang as an off-the-shelf front-end. This conversion removes high-level information about the structure of C++ programs (e.g., the relationship between classes). However, the code fragments that use the STL are inlined, which simplifies the verification process. From the LLVM intermediate representation, LLBMC generates a quantifier-free logical formula based on bit-vectors. This formula is further simplified and passed to an SMT solver for verification. The tool does not verify programs with exception handling, making it challenging to verify C++ programs realistically since exceptions must be disabled during the generation of the LLVM intermediate representation. The biggest difference between the tool described by the authors and the purpose of this work is related to the beginning of the verification process. In LLBMC, the conversion of the source program into an intermediate representation LLVM is required. The biggest obstacle to this approach is the need for a constant tool adjustment to new versions of the LLVM intermediate representation that the clang generates. For instance, a symbolic virtual machine built on top of the LLVM compiler, named as KLEE [68], still uses an old version of LLVM (v3.4) due to the significant effort to update its internal structure.

Developed by NASA, the MCP Model Checker [11] is yet another model checking tool based on the LLVM infrastructure specifically design to C++ programs. Authors claim the tool has full support for the C++ language; however, the tool is not publicly available and could not be included in our experimental evaluation. The tool does not extract a model from the source code, but it instruments the code with assertions through an LLVM-to-LLVM transformation. MCP then executes the code trying all possible interleaving in order to hit the injected assertions.

Blanc, Groce, and Kroening [14] describe the verification of C++ programs using containers via predicate abstraction. A simplified operational model using Hoare logic is proposed to support C++ programs that make use of the

STL. The purpose of the operational model is to simplify the verification process using the SATABS tool [69]. SATABS is a verification tool for C and C++ programs that supports classes, operator overloading, references, and templates (but without supporting partial specification). In order to verify the correctness of a program, the authors show that it is sufficient to use an operational model by proving that, if the pre- and post-conditions hold, the implementation model also holds. The approach is efficient in finding trivial errors in C++ programs. The preconditions are modelled to verify the library containers using an operational model similar to the ESBMC tool's model for the same purpose. Regarding the operational model, the authors present only preconditions. In contrast, our operational model verifies preconditions and replicates the STL behaviour, which increases the range of applications that can be adequately verified by the tool (i.e., postconditions).

Clarke, Kroening, and Lerda [25] present CBMC, which implements BMC for C/C++ programs using SAT/SMT solvers. CBMC uses its parser, based on Flex/Bison [20], to build an AST. The type-checker of CBMC's front-end annotates this AST with types and generates a language-independent intermediate representation of the original source code. The intermediate representation is then converted into an equivalent GOTO-program (i.e., control-flow graphs) that the symbolic execution engine will process. ESBMC improves the front-end, the GOTO conversion and the symbolic execution engine to handle the C++03 standard. CBMC and ESBMC use two 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 arithmetic overflow and underflow, array bounds violations, and null pointer dereferences, in the spirit of sites' clean termination [70]. 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 semantics of the program. A VC generator (VCG) then derives the verification conditions from them. CBMC is a well-known model checker for C programs, but its support for C++ is rather incomplete (cf. Section 6). In particular, CBMC has problems instantiating template correctly and lacks support for STL, exception specialization and terminate/unexpected functions.

Baranová et al. [16] present DIVINE, an explicit-state model checker to verify single- and multi-threaded programs written in C/C++ (and other input formats, such as UPPAAL[11] and DVE[12]). Another language supported by DIVINE is the LLVM intermediate representation; for this reason, the base of its verification process is the translation of C++ programs into that representation. Using clang as front-end, DIVINE translates C++ programs into the LLVM intermediate representation, thereby, applying its implementation of the C and C++ standard libraries in order to ensure a consistent translation. Nonetheless, this translation process might cause some irregularities to the verification process once it loses high-level information about the C++ program structure (i.e., the relationship between the classes). To tackle such issues in the verification process of exception handling structures, Štill, Ročkai and Barnat [67] propose a new API for DIVINE to properly map and deal with exception handling in C++ programs, based on a study about the C++ and LLVM exception handling mechanisms [66]. The authors also claim DIVINE as the first model checker that can verify exception handling in C++ programs, as opposed to what has been stated by Ramalho et al. [24]. However, ESBMC *v*1.23 (i.e., the version used by Ramalho et al. [24]) is able to correctly verify the example presented by Ročkai, Barnat and Brim [67], generating and verifying 10 VCs in less than 1 s. Our experimental evaluation shows that ESBMC outperforms DIVINE in handling exceptions as well as for the support of standard containers, inheritance, and polymorphism (cf. Section 6).

8 | CONCLUSIONS AND FUTURE WORK

We described a novel SMT-based BMC approach to verify C++ programs using ESBMC. We started with an overview of ESBMC's type-checking engine, which includes our approach to support templates (similar to conventional compilers) that replaces the instantiated templates before the encoding phase. We also describe our type-checking mechanism to handle single and multiple inheritance and polymorphism in C++ programs. We then present the significant contributions of this work: the C++ operational models (COM) and the support for exception handling. We describe an abstraction of the STL, which replaces them during the verification process. The purpose is twofold: reduce complexity while checking whether a given program uses the STL correctly. Finally, we present novel approaches to handle critical features of exception handling in C++ (e.g., unexpected and termination function handlers).

To evaluate our approach, we extended our previous experimental evaluation [24] by approximately 36%. ESBMC is able to verify correctly 84.27%, in approximately 4 h, outperforming two state-of-art verifiers, DIVINE and LLBMC (cf. Section 6). ESBMC and DIVINE were also able to verify programs with exceptions enabled, a missing feature of LLBMC that decreases the verification accuracy of real-world C++ programs. ESBMC was able

[11]https://www.uppaal.org
[12]https://divine.fi.muni.cz/index.html

to find undiscovered bugs in the *Sniffer* code, a commercial application of medium-size used in the telecommunications domain. The developers later confirmed the respective bugs. LLBMC was able to discover a subset of the bugs discovered by ESBMC, while DIVINE was unable to verify the application due to a lack of support for the Boost C++ library [71].

Our verification method depends on the fact that COM correctly represents the original STL. Indeed, the correctness of such a model to trust in the verification results is a significant concern [18,72-76]. The STL is specified by the ISO International Standard ISO/IEC 14882:2003(E) – Programming Language C++ [45]. Similar to conformance testing [77,78], to certify the correlation between STL and COM, we rely on the translation of the specification into assertions, which represents the pre- and post-conditions of each method/function in the SCL. Although COM is an entirely new implementation, it consists in (reliably) building a simplified model of the related STL, using the C/C++ programming language through the ESBMC intrinsic functions (e.g., **`assert`** and **`assume`**) and the original specification, which thus tends to reduce the number of programming errors. Besides, Cordeiro *et al.* [20,79,80] presented the soundness for such intrinsic functions already supported by ESBMC. Although proofs regarding the soundness of the entire operational model could be carried out, it represents a laborious task due to the (adopted) memory model [81]. Conformance testing concerning operational models would be a suitable approach [18,78] and represents a promising research direction.

For future work, we intend to extend ESBMC coverage in order to verify C++11 programs. The new standard is a huge improvement over the C++03, which includes the replacement of exception specialization by a new keyword **`noexcept`**, which works in the same fashion as an empty exception specialization. The standard also presents new sequential containers (**`array`** and **`forward_list`**), new unordered associative containers (**`unordered_set`**, **`unordered_multiset`**, **`unordered_map`** and **`unordered_multimap`**), and new multithreaded libraries (e.g., **`thread`**) in which our COM does not yet support. Finally, we will develop a conformance testing procedure to ensure that our COM conservatively approximates the STL semantics.

Furthermore, we intend to improve the general verification of C++ programs, including improved support for templates. Although the current support of templates was sufficient to verify real-world C++ applications (cf. Section 6) it is still work-in-progress. For instance, the handling of SFINAE [45] in ESBMC is limited, and limitations on the support of nested templates, as shown in the experiments, directly affect the verification process. This limitation is because template instantiation is notoriously hard, especially if we consider recent standards. Although our front-end can handle many real-world C++ programs, maintaining the C++ front-end in ESBMC is a herculean task. For that reason, we decided to rewrite our front-end using clang to generate the program AST. Importantly, we do not intend to use the LLVM intermediate representation but the AST generated by clang. In particular, if we use clang to generate the AST, then it solves several problems: (i) the AST generated by clang contains all the instantiated templates so we only need to convert the instantiated classes/functions and ignore the generic version; (ii) supporting new features will be as easy as adding a new AST conversion node from the clang representation to ESBMC representation; (iii) we do not need to maintain a full C++ front-end since ESBMC will contain all libraries from clang. Thus, we can focus on the main goal of ESBMC, the SMT encoding of C/C++ programs.

We already took the first step towards that direction and rewrote the C front-end [22], and the C++ front-end is currently under development.

ACKNOWLEDGEMENTS

The work in this paper is partially funded by the EPSRC grants EP/T026995/1, EP/V000497/1, EU H2020 ELEGANT 957286, Nokia Institute of Technology (INdT), and Soteria project awarded by the UK Research and Innovation for the Digital Security by Design (DSbD) Programme.

DATA AVAILABILITY STATEMENT

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

ORCID

Felipe R. Monteiro https://orcid.org/0000-0001-9420-9056
Mikhail R. Gadelha https://orcid.org/0000-0001-6540-6587
Lucas C. Cordeiro https://orcid.org/0000-0002-6235-4272

REFERENCES

1. Chong N, Cook B, Kallas K, Khazem K, Monteiro FR, Schwartz-Narbonne D, et al. Code-level model checking in the software development workflow. In *42nd International Conference on Software Engineering (ICSE)*: Seoul, Korea (South), 2020.
2. Szekeres L, Payer M, Wei T, Song D. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy*: Berkeley, CA, USA, 2013; 48–62.

3. Miller M. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. In Technical Report, Microsoft Security Response Center, 2019.
4. Hathhorn C, Rosu G. Dealing with C's original sin. IEEE Softw. 2019;36:24–8.
5. Clarke EM, Henzinger TA, Veith H. Handbook of model checking. Springer International Publishing, 2018; p.1–26.
6. Cook B, Khazem K, Kroening D, Tasiran S, Tautschnig M, Tuttle MR. 2018. Model checking boot code from AWS data centers. In Computer aided verification; 467–86.
7. Distefano D, Fahndrich M, Logozzo F, O'Hearn PW. Scaling static analyses at Facebook. Commun ACM. 2019;62:62–70.
8. Sadowski C, Aftandilian E, Eagle A, Miller-Cushon L, Jaspan C. Lessons from building static analysis tools at Google. Commun ACM. 2018; 61:58–66.
9. Chong N, Cook B, Eidelman J, Kallas K, Khazem K, Monteiro FR, et al. Code-level model checking in the software development workflow at Amazon web services. Softw Pract Exper. 2021;51(4):772–97.
10. Beyer D. 2019. Automatic verification of C and java programs: SV-COMP 2019. In Tools and algorithms for the construction and analysis of systems, LNCS, vol. 11429; 133–55.
11. Thompson S, Brat G. Verification of C++ flight software with the MCP model checker. In *2008 IEEE Aerospace Conference*: Big Sky, MT, USA, 2008; 1–9.
12. Merz F, Falke S, Sinz C. 2012. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In Verified software: Theories, tools, and experiments, LNCS, vol. 7152; 146–61.
13. Yang J, Balakrishnan G, Maeda N, Ivančič F, Gupta A, Sinha N, et al. 2012. Object model construction for inheritance in C++ and its applications to program analysis. In Compiler construction, LNCS, vol. 7210; 144–64.
14. Blanc N, Groce A, Kroening D. 2007. Verifying C++ with STL containers via predicate abstraction. In Automated software engineering; 521–4.
15. Prabhu P, Maeda N, Balakrishnan G. Interprocedural exception analysis for C++. In *European conference on object-oriented programming*, LNCS, vol. 6813, 2011; 583–608.
16. Baranová Z, Barnat J, Kejstová K, Kučera T, Lauko H, Mrázek J, et al. 2017. Model checking of C and C++ with DIVINE 4. In Automated technology for verification and analysis, LNCS, vol. 10482; 201–7.
17. Carter M, He S, Whitaker J, Rakamarić Z, Emmi M. SMACK software verification toolchain. In *International Conference on Software Engineering*, 2016; 589–92.
18. Monteiro FR, Garcia MAP, Cordeiro LC, de Lima Filho EB. Bounded model checking of C++ programs based on the Qt cross-platform framework. Softw Test Verif Reliab. 2017;27(3): 24.
19. Biere A, Heule M, van Maaren H, Walsh T. Handbook of satisfiability: Volume 185 frontiers in artificial intelligence and applications (1st edn.), vol. 185. IOS Press, 2009.
20. Cordeiro LC, Fischer B, Marques-Silva J. SMT-based bounded model checking for embedded ANSI-C software. IEEE Trans Softw Eng. 2012; 38(4):957–74.
21. Morse J, Cordeiro LC, Nicole DA, Fischer B. Applying symbolic bounded model checking to the 2012 RERS greybox challenge. Softw Tools Technol Trans. 2014;16(5):519–29.
22. Gadelha MR, Monteiro FR, Morse J, Cordeiro LC, Fischer B, Nicole DA. 2018. ESBMC 5.0: An industrial-strength C model checker. In Automated software engineering; 888–91.
23. Gadelha MR, Monteiro F, Cordeiro L, Nicole D. 2019. ESBMC v6.0: Verifying C programs using k-induction and invariant inference. In Tools and algorithms for the construction and analysis of systems, LNCS, vol. 11429; 209–13.
24. Ramalho M, Freitas M, Sousa F, Marques H, Cordeiro LC, Fischer B. 2013. SMT-based bounded model checking of C++ programs. In Engineering of computer based system; 147–56.
25. Clarke E, Kroening D, Lerda F. 2004. A tool for checking ANSI-C programs. In Tools and algorithms for the construction and analysis of systems, LNCS, vol. 2988; 168–76.
26. Niemetz A, Preiner M, Biere A. Boolector 2.0. J Satisfiab, Boolean Model Comput. 2014;9:53–8.
27. De Moura L, Bjørner N. 2008. Z3: An efficient SMT solver. In Tools and algorithms for the construction and analysis of systems, LNCS, vol. 4963; 337–40.
28. Dutertre B. 2014. Yices 2.2. In Computer-aided verification, LNCS, vol. 8559; 737–44.
29. Cimatti A, Griggio A, Schaafsma B, Sebastiani R. 2013. The mathSAT5 SMT solver. In Tools and algorithms for the construction and analysis of systems, LNCS, vol. 7795; 93–107.
30. Barrett C, Conway C, Deters M, Hadarean L, Jovanović D, King T, et al. 2011. CVC4. In Computer-aided verification, LNCS, vol. 6806; 171–7.
31. Beyer D. 2020. Advances in automatic software verification: SV-COMP 2020. In Tools and algorithms for the construction and analysis of systems, LNCS, vol. 12079; 347–67.
32. Beyer D. 2020. Second competition on software testing: Test-comp 2020. In Fundamental approaches to software engineering, LNCS, vol. 12076; 505–19.
33. Abreu RB, Gadelha MR, Cordeiro LC, Filho EBL, da Silva Jr. WS. Bounded model checking for fixed-point digital filters. J Brazilian Comput Soc. 2016;22(1):1:1–20.
34. de Bessa IV, Ismail HI, Cordeiro LC, Filho JEC. Verification of delta form realization in fixed-point digital controllers using bounded model checking. In *Brazilian Symposium on Computing Systems Engineering*: Manaus, Brazil, 2014; 49–54.
35. Chaves LC, Bessa I, Ismail H, dos Santos Frutuoso AB, Cordeiro LC, de Lima Filho EB. DSVerifier-aided verification applied to attitude control software in unmanned aerial vehicles. IEEE Trans Reliab. 2018;67(4):1420–41.
36. Muchnick SS. Advanced compiler design and implementation (1st edn.) Morgan Kaufmann Publishers Inc., 1997.
37. Kroening D, Ouaknine J, Strichman O, Wahl T, Worrell J. 2011. Linear completeness thresholds for bounded model checking. In Computer-aided verification, LNCS, vol. 6806; 557–72.
38. Bradley AR, Manna Z. The calculus of computation: Decision procedures with applications to verification(1st edn.) Springer-Verlag: New York, Inc., 2007.
39. McCarthy J. Program verification: Fundamental issues in computer science. Springer Netherlands, 1993; p.35–56.
40. Preiner M, Niemetz A, Biere A. Better lemmas with lambda extraction. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD'15: Austin, TX, USA, 2015; 128–35.

41. Brummayer R, Biere A. 2009. Boolector: An efficient SMT solver for bit-vectors and arrays. In Tools and algorithms for the construction and analysis of systems, LNCS, vol. 5505 Springer: Berlin, Heidelberg; 174–7.
42. GCC, the GNU compiler collection, 2015. https://gcc.gnu.org/. [Online; accessed August-2019].
43. Stroustrup B. The C++ programming language (3rd edn.) Addison-Wesley Longman Publishing Co., Inc., 2000.
44. Cordeiro LC, Fischer B, Marques-Silva J. Continuous verification of large embedded software using SMT-based bounded model checking, 2010; 160–9.
45. ISO. C++ standard, 2003. ISO/IEC 14882:2003.
46. Siek J, Taha W. 2006. A semantic analysis of C++ templates. In European conference on object-oriented programming, LNCS, vol. 4067 Springer: Berlin, Heidelberg; 304–27.
47. Deitel HM, Deitel PJ. C++ how to program (6th edn.) Prentice Hall Press, 2007.
48. Pasareanu CS, Visser W. 2004. Verification of java programs using symbolic execution and invariant generation. In Model checking of software, LNCS, vol. 2989 Springer: Berlin, Heidelberg; 164–81.
49. Anand S, Păsăreanu CS, Visser W. 2007. JPF-SE: A symbolic execution extension to java pathfinder. In Tools and algorithms for the construction and analysis of systems, LNCS, vol. 4424; 134–8.
50. Ramananandro T, Dos Reis G, Leroy X. Formal verification of object layout for C++ multiple inheritance. In *Symposium on Principles of Programming Languages*, 2011; 67–80.
51. Neggers J, Kim H. Basic posets (1st edn.) World Scientific, 1999.
52. Alexander RT, Offutt J, Bieman JM. Fault detection capabilities of coupling-based OO testing. In *13th International Symposium on Software Reliability Engineering, 2002. Proceedings*: Annapolis, MD, USA, 2002; 207–2.
53. Driesen K, Hölzle U. The direct cost of virtual function calls in C++. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1996; 306–23.
54. Kroening D, Tautschnig M. CBMC - C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 8413, 2014; 389–91.
55. Morse J, Ramalho M, Cordeiro LC, Nicole D, Fischer B. ESBMC 1.22. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 8413, 2014; 405–7.
56. Dos Reis G, García JD, Logozzo F, Fahndrich M, Lahiri S. Simple contracts for C++, 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0287r0.pdf. [Online; accessed May-2020].
57. Verdi M, Sami A, Akhondali J, Khomh F, Uddin G, Motlagh AK. An empirical study of C++ vulnerabilities in crowd-sourced code examples, 2019.
58. Monteiro FR, Gadelha MR, Cordeiro LC. Model checking C++ programs—Replication package, 2021. https://doi.org/10.5281/zenodo.4579853. [Online; accessed March-2021].
59. Reference of the C++ language library, 2013. http://www.cplusplus.com. [Online; accessed August-2019].
60. Xusheng X, Gogul B, Franjo I, Naoto M, Aarti G. NECLA benchmarks: C++ programs with C++ specific bugs, 2013. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.410.7773&rep=rep1&type=pdf. [Online; accessed August-2019].
61. Beyer D. Software verification and verifiable witnesses—(report on SV-COMP 2015). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 9035, 2015; 401–16.
62. Morneault K, Pastor-Balbas J. Signaling system 7 (SS7) message transfer part 3 (MTP3)-user adaptation layer (m3UA). In Technical Report, The Internet Society, 2006.
63. Gurfinkel A, Kahsai T, Komuravelli A, Navas JA. The seahorn verification framework. In *International Conference on Computer Aided Verification*, vol. 9206, 2015; 343–61.
64. Barnett M, Chang B-YE, DeLine R, Jacobs B, Leino KRM. Boogie: A modular reusable verifier for object-oriented programs. In *Symposium on formal methods for components and objects*, vol. 4111, 2006; 364–87.
65. Falke S, Merz F, Sinz C. The bounded model checker LLBMC. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013; 706–9.
66. Ročkai P, Barnat J, Brim L. Model checking C++ programs with exceptions. Sci Comput Programm. 2016;128:68–85.
67. Štill V, Ročkai P, Barnat J. Using off-the-shelf exception support components in C++ verification. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*: Prague, Czech Republic, 2017; 54–64.
68. Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on operating systems design and implementation*, 2008; 209–24.
69. Clarke E, Kroening D, Sharygina N, Yorav K. SATABS: SAT-based predicate abstraction for ANSI-C. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 3440, 2005; 570–4.
70. Sites RL. Some thoughts on proving clean termination of programs. In Technical Report, Computer Science Department, Stanford University, 1974.
71. Boost C++ libraries documentation, 2005. http://www.boost.org/doc/. [Online; accessed August-2019].
72. van der Merwe H, Tkachuk O, van der Merwe B, Visser W. Generation of library models for verification of android applications. Softw Eng Notes. 2015;40(1):1–5.
73. Monteiro FR, Cordeiro LC, de Lima Filho EB. Bounded model checking of C++ programs based on the Qt framework. In *Global Conference on Consumer Electronics*, 2015; 179–80.
74. Pereira P, Albuquerque H, da Silva I, Marques H, Monteiro FR, Ferreira R, et al. SMT-based context-bounded model checking for CUDA programs. Concurrency Comput Pract Exper. 2017;29(22):e3934.
75. Garcia M, Monteiro FR, Cordeiro LC, de Lima Filho EB. ESBMC^{QtOM}: A bounded model checking tool to verify qt applications. In *SPIN*, LNCS, vol. 9641, 2016; 97–103.
76. Monteiro FR, Alves EHS, Silva IS, Ismail HI, Cordeiro LC, Filho EBL. ESBMC-GPU a context-bounded model checking tool to verify CUDA programs. Sci Comput Programm. 2018;152:63–9.
77. de la Cámara P, Gallardo M-M, Merino P, Sanán D. Checking the reliability of socket based communication software. Softw Tools Technol Trans. 2009;11(5):359–74.
78. Cámara P, Castro JR, Gallardo M-M, Merino P. Verification support for ARINC-653-based avionics software. Softw Test Verif Reliab. 2011; 21(4):267–98.

79. Cordeiro LC, Fischer B. Verifying multi-threaded software using SMT-based context-bounded model checking. In *International Conference on Software Engineering*: Honolulu, HI, USA, 2011; 331–40.
80. Cordeiro LC. SMT-based bounded model checking for multi-threaded software in embedded systems. In *International Conference on Software Engineering*: Cape Town, South Africa, 2010; 373–6.
81. Mehta F, Nipkow T. Proving pointer programs in higher-order logic. Inform Comput. 2005;199(1):200–27.

How to cite this article: Monteiro FR, Gadelha MR, Cordeiro LC. Model checking C++ programs. Softw Test Verif Reliab. 2022;32:e1793. https://doi.org/10.1002/stvr.1793