ESBMC v7.6: Enhanced Model Checking of C++ Programs with Clang AST

Xianzhiyu Li[a,*], Kunjian Song[a], Mikhail R. Gadelha[b], Franz Brauße[a], Rafael S. Menezes[a,c], Konstantin Korovin[a], Lucas C. Cordeiro[a,c]

[a] *The University of Manchester, Oxford Rd, Manchester, M13 9PL, England, UK*
[b] *Igalia, Bugallal Marchesi, 22, 1º, A Coruña, 15008, Galicia, Spain*
[c] *Federal University of Amazonas, Av. General Rodrigo Octávio Jordão Ramos, 6200, Manaus, 69080-005, Amazonas, Brazil*

Abstract

This paper presents Efficient SMT-Based Context-Bounded Model Checker (ESBMC) v7.6, an extended version based on previous work on ESBMC v7.3 by K. Song et al. [1]. The v7.3 introduced a new Clang-based C++ front-end to address the challenges posed by modern C++ programs. Although the new front-end has demonstrated significant potential in previous studies, it remains in the developmental stage and lacks several essential features. ESBMC v7.6 further enhanced this foundation by adding and extending features based on the Clang AST, such as ① exception handling, ② extended memory management and memory safety verification, including dangling pointers, duplicate deallocation, memory leaks and rvalue references and ③ new operational models for STL updating the outdated C++ operational models. Our extensive experiments demonstrate that ESBMC v7.6 can handle a significantly broader range of C++ features introduced in recent versions of the C++ standard.

Keywords: Formal Methods, Model Checking, Software Verification

*Corresponding author

Email addresses: xianzhiyu.li@postgrad.manchester.ac.uk (Xianzhiyu Li), kunjian.song@postgrad.manchester.ac.uk (Kunjian Song), mikhail@igalia.com (Mikhail R. Gadelha), franz.brausse@manchester.ac.uk (Franz Brauße), rafael.menezes@postgrad.manchester.ac.uk (Rafael S. Menezes), konstantin.korovin@manchester.ac.uk (Konstantin Korovin), lucas.cordeiro@manchester.ac.uk (Lucas C. Cordeiro)

1. Introduction

C++ is one of the most popular programming languages used to build high-performance and real-time systems, such as operating systems, banking systems, communication systems, and embedded systems [2]. However, memory safety issues remain a major source of security vulnerabilities in C++ programs [3]. Fan et al. [4] created a dataset of C/C++ vulnerabilities by mining the Common Vulnerabilities and Exposures (CVE) database [5] and the associated open-source projects on GitHub, then curated the issues based on Common Weakness Enumeration (CWE) [6]. According to their findings, two out of the top three vulnerabilities are caused by memory safety issues: Improper Restriction of Operations within the Bounds of a Memory Buffer (CWE-119) and Out-of-bounds Read (CWE-125) [4].

The limitation of software testing resides in the user inputs [7]. Only a limited number of execution paths may be tested since test cases involve human inputs in the form of concrete values [8]. In contrast to testing, formal verification techniques can be used more systematically to formally reason about a program, although they suffer from the state-space explosion problem [9]. There is an increasing adoption of formal verification techniques for C programs in the industry, e.g., Amazon has been using model-checking techniques to prove the correctness of their C-based systems in Amazon Web Services (AWS); this has positively impacted their code quality, as evidenced by the increased rate of bugs found and fixed [10].

Formal verification of C++ programs is more challenging than C programs due to the sophisticated features, such as the STL (Standard Template Libraries) containers, templates, exception handling, and object-oriented programming (OOP) paradigm [2]. Several tools have been developed for verification of C++ programs, most prominent are: CBMC [11], DIVINE [12], and ESBMC [1]. But the existing state-of-the-art verification tools have only limited C++ feature support [13]. CBMC [11] is a bounded model checker that converts source code into an intermediate representation for verification, with limited support for polymorphism and exception handling [14]. DIVINE [12], an explicit-state model checker, uses LLVM bitcode [15] as an intermediate representation for verifying C++ programs, with limited ability to handle standard containers and inheritance [13]. For ESBMC, Ramalho et al. [16] and Monteiro et al. [13] initiated the support for C++ program verification. Since then, ESBMC has undergone heavy development to support recent versions of the C++ standard [17]. This research builds upon the Clang-based C++

front-end introduced in ESBMC v7.3, which replaced the legacy parser with a more robust tool. In this work, we refine and extend these contributions:

- **Complete Redesign**: ESBMC's C++ front-end has undergone a complete restructuring and now relies on Clang [15]. By leveraging Clang's parsing and semantic analysis capabilities [18, 19], we check the input program's Abstract Syntax Tree (AST) using a production-quality compiler. This eliminates static analysis logic and ensures enhanced accuracy and efficiency.
- **Object Models Details**: We provide comprehensive insights into the object models used to achieve seamless conversion of C++ polymorphism code to ESBMC's Intermediate Representation (IR).
- **Simplified Type Checking for Templates:** The new Clang-based front-end greatly simplifies type checking for templates, streamlining ESBMC's ability to adapt to C++ advancements.

Between ESBMC v7.3 and this work, we not only refined existing features but also introduced several new verification capabilities:

- **Extended C++ Memory Management:** We have extended the implementation of the dynamic memory operators `new` and `delete` in our new front-end, which enhances ESBMC's ability to verify memory safety issues.
- **Modeled Rvalue References:** Our new Clang-based C++ front-end models the key C++11 feature of *rvalue* references [20], supports the *move* function and *move* semantics.
- **C++ Exception Handling:** We have implemented exception handling based on Clang AST and extended the exception specification. Additionally, we adapted the symbolic engine to match thrown exceptions. This enhancement enables ESBMC to support exceptions in C++11 and later versions.
- **Updated C++ Operational Models (OMs):** We enabled OMs from ESBMC v2.1 and maintained the outdated OMs to adapt them for the new Clang-C++ front-end.

By introducing these features, our work significantly enhances ESBMC's C++ verification capabilities, paving the way for more robust and efficient verification of C++ programs and their variants. This work primarily focuses on the improvements made to ESBMC across its different versions, with contributions being relative to its previous iterations. Comparisons with other verification tools will be explored in future work.

This paper is organized as follows: we begin with a brief introduction to SMT-based BMC techniques and the limitations of previous versions of ESBMC. In Section 3, we present the implementation of core C++ language features based on the Clang AST, along with detailed process flows. Section 4 provides the experimental results of the benchmarks used and analyzes potential threats to validity. Finally, in Section 5, we conclude and outline future work.

2. Background

ESBMC's verification for C++03 programs reached its maturity in version v2.1, presented by Monteiro et al. [13]. ESBMC v2.1 provides a first-order logic-based framework that formalizes a wide range of C++ core languages, verifying the input C++ programs by encoding them into SMT formulas. Since C++ Standard Template Libraries (STL) contain optimized assembly code not verifiable using ESBMC, ESBMC v2.1 tackled this problem using a collection of C++ Operational Models (OMs) to replace the STL included in the input program. The OMs are abstract representations mimicking the structure of the STL, adding pre- and post-conditions to all STL APIs [21]. Combining these approaches, ESBMC v2.1 outperformed other state-of-the-art tools evaluated over a large set of benchmarks, comprising 1513 test cases [13]. Nonetheless, ESBMC v2.1 employs a Flex and Bison-based front-end from CBMC [11], which leads to hard-to-maintain code and can hardly evolve to support modern features introduced in C++11 and later versions. Between v2.1 and v7.3, the focus was on developing a new C++ front-end. C++ verification using Clang AST was introduced in v7.3, which significantly improved the Clang-based front-end and made it the first version capable of handling most modern C++ features. As a result, v2.1 serves as the meaningful point of comparison, as it represents the state of ESBMC before the new C++ front-end was introduced.

2.1. SMT-based BMC technique

The core functionality of ESBMC is based on SMT solvers to process a decidable fragment of first-order logical formulas derived from intermediate representations (IR), thereby enabling efficient model checking. In BMC, the analyzed program is modeled as a state transition system, derived from the control-flow graph (CFG) [22]. The CFG is created during the translation from program code to single static assignment (SSA) form. Nodes in the CFG represent assignments or conditional statements, while edges represent possible changes in the program's control flow. Consider a transition system M, a property ϕ, and an integer parameter k. Bounded Model Checking (BMC) unrolls the system k times, converting it into a verification condition ψ_k. The condition ψ_k is satisfiable if and only if a counterexample of length k or less exists for the property ϕ. This model-checking problem can be formalized by constructing the following logical formula:

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

In this formulation, I denotes the set of initial states of M, $T(s_i, s_{i+1})$ describes the transition relation between states s_i and s_{i+1}, and $\phi(s_i)$ is the safety property evaluated at state s_i. The formula $I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$ represents the executions of M over k steps, while $\bigvee_{i=0}^{k} \neg\phi(s_i)$ indicates that ϕ is violated in some state s_i for $0 \leq i \leq k$. In cases where the formula (1) is satisfiable, an SMT solver can produce a satisfying assignment. This assignment enables us to determine the values of the program variables, which can then be used to construct a counterexample. Such a counterexample for the property ϕ is a sequence of states $s_0, s_1, \ldots, s_k$ where $s_0 \in S_0$, with S_0 representing the set of initial states, and $T(s_i, s_{i+1})$ holds for $0 \leq i < k$.

On the other hand, if the formula (1) is unsatisfiable, no error state is reachable in k steps or fewer. However, this does not guarantee the completeness of BMC techniques, as counterexamples with lengths exceeding k may still exist. To ensure completeness, it is necessary to establish an upper bound on the depth of the state space. This involves confirming that all significant system behaviors have been explored, ensuring that further search only exhibits states that have already been verified, which can be achieved using k-induction to prove that the system remains correct beyond the explored states [23, 24].

2.2. Old C++ front-end in v2.1

The version of ESBMC in Monteiro et al. [13] used an outdated CPROVER-based front-end [11] with the following limitations.

1. For the type-checking phase, ESBMC could not provide meaningful warnings or error messages.
2. It was inefficient at generating a body for default implicit non-trivial methods in a class, such as C++ copy constructors or copy assignment operators.
3. The parser of the old front-end needed to be manually updated to cover the essential C++ semantic rules [25], which leads to hard-to-maintain code to keep up with the C++ evolution.
4. The old front-end contained excessive data structures and procedures auxiliary to scope resolution and function type-checking.
5. The type-checker [25] of the old front-end only worked with a CPROVER-based parse tree and supports features up to the C++03 standard [26]. We found adapting it to the new C++ language and library features difficult.
6. The old front-end used a speculative approach to guess the arguments for a template specialization and a map to associate the template parameters to their instantiated values, which leads to hard-to-maintain and hard-to-debug code in the case of recursive templates. Additionally, owing to its limited static analysis, the old front-end could not provide any early warning when there is a circular dependency between the templates.

Following the introduction of ESBMC v7.3 by K. Song et al. [1], these limitations have been addressed. Clang can provide detailed error messages and warnings during the type-checking phase and automatically generate implicit methods for classes. Notably, the Clang-based approach solves the problem of needing continuous maintenance to adapt to new C++ feature changes. However, ESBMC v7.3 lacked full support for *rvlaue* references, exception handling, and dynamic memory verification. These limitations affected its ability to handle modern C++ programs. In this work, we extend the Clang-based C++ front-end by addressing these gaps, proposing approaches to ensure compliance with recent C++ standards [17].

3. Model Checking C++ Programs using Clang AST

Figure 1 illustrates ESBMC's verification pipeline for C++ programs. The new Clang-C++ front-end type-checks and converts the input C++ program (along with the corresponding OMs) into the GOTO program representation [14, 27]. Then, the GOTO program will be symbolically executed to generate the SSA form of the program, thus generating a set of logical formulas consisting of the constraints and properties. An SMT solver is used to check the satisfiability of the formulas, giving a verdict *VERFICATION SUCCESSFUL* if no property violation is found up to the bound k or a counterexample in case of property violation (cf. Section 2.1).

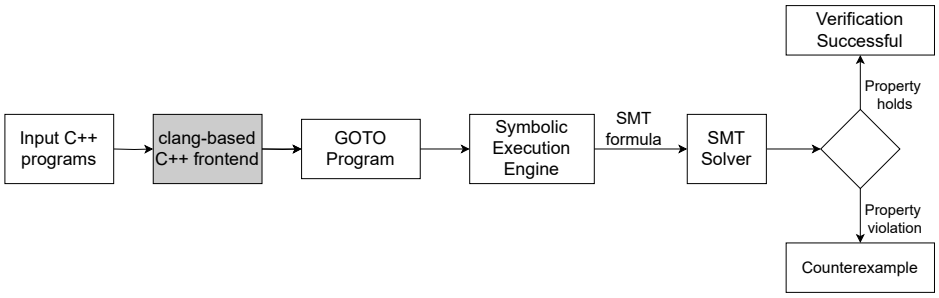


Figure 1: ESBMC architecture for C++ verification. The grey block represents the extended Clang-based C++ front-end developed in ESBMC v7.6.

3.1. Polymorphism

The traditional approach for achieving polymorphism makes use of virtual function tables (also known as *vtables*) and virtual pointers (known as *vptrs*). While the Clang AST does not include information about virtual tables or virtual pointers of a class, it provides users with enough information to enable them to create their own *vtables* and *vptrs*. In the new Clang-based C++ front-end, we reimplemented the *vtable* and *vptr* construction mechanism following a similar approach as ESBMC v2.1, but with significant simplifications based on the information provided in the Clang AST. Figure 2 illustrates an example of C++ polymorphism.

Figure 3 illustrates the object models for the example classes `Bird` and `Penguin`. The new front-end adds one or more *vptrs* to each class. The *vptrs* will be initialized in the class constructors, which set each *vptr* pointing to the desired *vtable*. The child class contains an additional pointer pointing to

```
1  class Bird {
2    public:
3    virtual int doit(void) { return 21; }
4  };
5
6  class Penguin: public Bird {
7    public:
8    int doit(void) override { return 42; }
9  };
10
11 int main() {
12   Bird *p = new Penguin();
13   assert(p->doit() == 42);
14   delete p;
15   return 0;
16 }
```

Figure 2: Example of C++ classes with virtual functions.

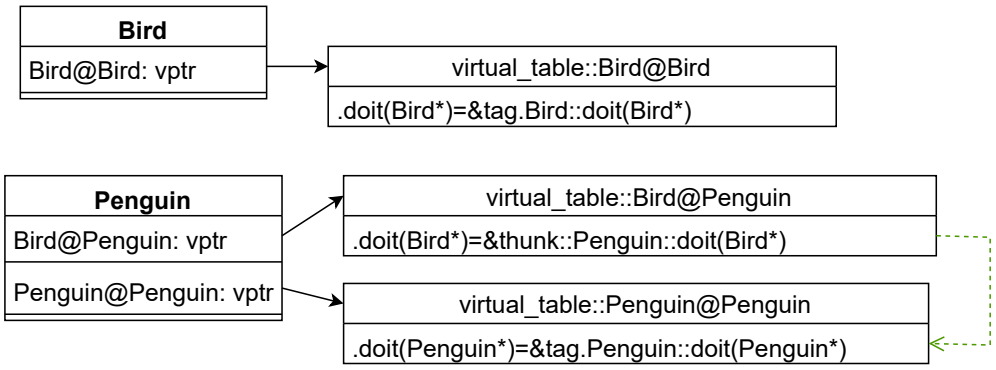


Figure 3: Object models for Bird and Penguin classes

```
1 int return_value;
2 return_value =
3 *p->Bird@Penguin
4     ->doit(p)
5 assert(return_value == 42)
```

(a) GOTO program of the dynamic dispatch in Line 12 of Figure 2.

```
1  thunk::Penguin::doit(Bird*):
2    int return_value;
3    return_value =
4      Penguin::doit(
5        (Penguin*)this)
6    RETURN: return_value
7    END_FUNCTION
8
9  Penguin::doit(Penguin*):
10   RETURN: 42
11   END_FUNCTION
```

(b) Thunk redirecting the call to the overriding function.

Figure 4: GOTO conversions of the overriding methods and dynamic dispatch.

a *vtable* with a thunk to the overriding function. The thunk redirects the call to the corresponding overriding function. In case of multiple inheritances, the child class would have multiple *vtprs* "inherited" from multiple base classes. The new front-end can also manage virtual inheritance, such as the diamond problem, which avoids duplicating *vptrs*, referring to the same virtual table in an inheritance hierarchy. Lines 2-4 in Figure 4a illustrate how dynamic dispatch is achieved using the *vptr* calling the thunk, which in turn calls the desired overriding function in Figure 4b, lines 9-11. Note that the *override* specifier is a C++11 extension that the old front-end could not support. As shown in Process Flow 1, class definitions are parsed from Clang AST to establish derived and base class relationships. An object model is created with *vptrs* to the *vtable*. Method calls are redirected through *vtpr* to overriding functions, or the base class function is invoked if no override exists.

Process Flow 1 Polymorphism

1: Parse the class definition from Clang AST to get the derived class and its base class
2: Create an *object model* for classes, storing each *vtpr* that point to the *vtable*.
3: **if** Use *vtpr* call thunk virtual functions **then**
4: Redirect the call to the corresponding overriding function
5: **else**
6: Call the base class function
7: **end if**

3.2. Templates

Templates are a key feature in C++, allowing types and certain values to be passed as parameters to types, values and functions. Templates allow STL containers and generic algorithms to work with different C++ data types [28, 29]. The old front-end in ESBMC v2.1 implements template specialization based on Siek et al. [30, 13]. However, it produces a "CONVERSION ERROR" for the test case illustrated in Figure 5a. This benchmark is based on the *Friend18* example from the GCC test suite [31], which was added for Bug 10158 on GCC Bugzilla [32]. ESBMC v7.6 successfully verified this benchmark and found the assertion's property violation in Figure 5a. The verification result is illustrated in Figure 5b. The example in Figure 5a contains a C++20 extension. The *foo* function is defined in *struct X* but gets called using an unqualified name with explicit template arguments in *main*. ESBMC v2.1 failed to verify it due to the "CONVERSION ERROR symbol 'foo' not found". We also tried this example with CBMC 5.88.1 [33] and cppcheck v2.11.1 [34], both current as of the time of writing. CBMC aborted during type-checking, while cppcheck did not provide any verification verdict. The Process Flow 2 shows how the Clang-based C++ front-end parses and processes templates.

```
1  #include <cassert>
2  template <int N> struct X
3  {
4    template <int M>
5    friend int foo(X const &)
6    {
7      return N * 10000 + M;
8    }
9  };
10 X<1234> bring;
11
12 int main() {
13   assert(
14    foo<5678> (bring)
15     !=12345678);
16 }
```

(a) Example of C++ class template

```
1  Violated property:
2    file tmp2.cpp
3     line 13 column 3
4      function main
5    assertion
6     foo<5678>(bring)
          !=12345678
7    return_value!=12345678
8
9  VERIFICATION FAILED
```

(b) Verdict for the template example

Figure 5: ESBMC verified the *Friend18* example from the GCC test suite. [31]

Process Flow 2 Templates

1: Parses Template Class definition and Friend Function Templates defined in it
2: Instantiate the Template Class and Friend Function Template
3: Convert Clang AST into an intermediate representation (IR)

3.3. C++ New and Delete

New and *Delete* are key operators for dynamic memory management in C++. *New* is used to dynamically allocate memory space and assign a pointer to that space to a variable, *Delete* is used to release the memory allocated by *New*. Note that this can lead to memory safety issues if used incorrectly [9]; for example, if a pointer is not set to *nullptr* after releasing the memory, it is a dangling pointer. When the program continues to use this dangling pointer, it may lead to undefined behavior. ESBMC v7.6 can verify such memory safety issues and the asserted property violation gives accurate information as shown in Figure 6b.

```
1  class Foo {
2    public:
3      Foo() {value = 0;};
4      void Inc() {value++;};
5    private:
6      int value;
7  };
8
9  int main() {
10   Foo *foo = new Foo();
11   delete foo;
12   foo->Inc();
13   return 0;
14 }
```

(a) Example of C++ new and delete

```
1  Violated property:
2   file main6.cpp
3    line 4 column 17
4     function Inc
5   dereference failure:
6    invalidated dynamic object
7
8  VERIFICATION FAILED
```

(b) Verdict for the new example

Figure 6: ESBMC verified an example with new and delete

ESBMC v7.6 employs a flat memory model and treats dynamically allocated memory as a contiguous region. It supports dynamic arrays and allows pointer arithmetic, with strict bounds checking to prevent out-of-bounds access while allowing unbounded array sizes. Built-in arrays track variables that point to dynamic memory, including their size and validity; each dynamic

object is marked as invalid when deallocated using *delete*. It ensures the validity of dynamic objects when accessed and modified, enabling it to verify a wide range of memory safety issues, such as duplicate deallocation and memory leaks. We also extend the implementation of *delete* to check whether the correct operator is used based on the size. For example, memory allocated for dynamic arrays should be released using the *delete* array operator, which is not supported in ESBMC v2.1 [13].

The Process Flow 3 begins by parsing the *new* and *delete* operators and the object from the Clang AST. The object is initialized, and a new dynamic object is created and marked as valid. When the *delete* operator is invoked, the memory allocation is verified for consistency, and the dynamic object is marked as invalid. During dereferencing operations, the validity of the dynamic object is checked to ensure safe access.

Process Flow 3 C++ *New* and *Delete*

1: Parse the *New*, *Delete* operators and object via Clang AST
2: Initialize the object (e.g., call the constructor)
3: Create a *dynamic object* for it and mark it as *valid*
4: **if** Call the *Delete* operator to free the memory allocated **then**
5: Check if the operator matches (e.g., new[] and delete[])
6: Mark the corresponding *dynamic object* as *invalid*
7: **end if**
8: **if** Dereference the *dynamic object* **then**
9: Check if the *dynamic object* is valid
10: **end if**

3.4. Rvalue References

Rvalue references were introduced in C++11, enhancing the efficiency of object operations and its syntax is to append double & after the type [20]. As the foundation of move semantics, it allows transferring resources from temporary objects to another object, eliminating the need for costly deep copy operations. This greatly enhances the efficiency of object assignment and passing, especially for large objects and dynamically allocated resources.

3.4.1. Move Semantics

ESBMC v7.6 models *rvalue* references and *move* semantics based on the information provided by Clang AST; we also supported the *move* function, which is used to convert the parameter into an *rvalue* reference explicitly. As shown in Process Flow 4, we model *rvalue* references as pointers, which

are dereferenced when assigned an *rvalue* or involved in computations with *rvalues*, including boolean operations. The example provided in Figure 7a involves the usage of the *move* function and the assignment to *rvalue* reference. Figure 7b illustrates our Clang-based C++ front-end's modeling of *rvalue* reference and the special treatment given to their operations. Assertions are used to ensure the validity of each step of these operations.

```
1  #include <cassert>
2  #include <utility>
3  int main() {
4      int a = 10;
5      int &&rref = std::move(a);
6      assert(rref == 10);
7      rref = 5;
8      assert(rref == 5);
9      return 0;
10 }
```

(a) Example of Rvalue reference

```
1  signed int a;
2  a = 10;
3  signed int *rref;
4  signed int *return_value;
5  FUNCTION_CALL:
6    return_value = move(&a)
7  rref = return_value;
8  assert(*rref == 10);
9  *rref = 5;
10 assert(*rref == 5);
```

(b) GOTO program of Rvalue reference

Figure 7: ESBMC verified an example with Rvalue reference

3.4.2. Move Member Functions

Move constructors and move assignment operators are the main applications of *rvalue* references; they are typically used in classes that manage resources, aiming to optimize resource movement and enhance efficiency. In Clang, when a C++ class or struct does not explicitly define the move semantics member functions, the compiler automatically generates them. The new C++ front-end of ESBMC v7.6 parses these default member functions from the Clang AST, while in ESBMC v2.1, undefined default member functions were not feasible. In line 9 of Figure 8a, we use the default *move* constructor to initialize the struct. Clang's AST provides it since we have not explicitly defined a constructor in the struct.

Process Flow 4 Rvalue References

1: Parse rvalue reference variable from Clang AST
2: Model the rvalue reference as a *pointer*
3: **if** Involves calculations with rvalues or assignments to rvalues **then**
4: Make special adjustment: dereference the *pointer*
5: **end if**

```
#include <cassert>
#include <utility>
struct MyStruct {
  int value;
};

int main() {
  MyStruct a{10};
  MyStruct b(std::move(a));
  assert(b.value == 10);
}
```

(a) Example of move member functions

```
MyStruct a;
a={ .value=10 };
MyStruct b;
struct MyStruct * return_value;
FUNCTION_CALL:
  return_value = move(&a)
FUNCTION_CALL:
  MyStruct(&b, return_value)
assert(b.value == 10);
```

(b) GOTO program of move member functions.

Figure 8: ESBMC verified an example with move member functions.

3.5. Exception Handling

Exception handling is a method that C++ uses to manage runtime errors [35]; it helps programs handle errors safely and prevent the program from crashing. This approach involves three main components: (1) the throw statement, which is used to raise an exception, (2) the try block, which contains the code that might throw an exception and directs to the first matching catch statement, and (3) the catch statement, which handles the exceptions raised by the throw statement. In our latest Clang-based C++ front-end, we have redesigned the exception handling mechanism, adopting a method similar to that used in ESBMC v2.1. In the new implementation, the front-end parses these components from the Clang AST, which enables better handling complex constructs, such as nested exceptions.

The GOTO program in Figure 9b illustrates how exception handling works. The first *catch* instruction marks the start of the *try* block. This instruction holds the tag assigned to each catch statement and the target location of their respective *catch* blocks. If an exception is thrown, ESBMC follows defined rules to jump to the appropriate *catch* statement, including potentially jumping to an invalid catch that triggers a verification error, indicating that the exception cannot be caught. If a suitable exception handler is found, the *thrown* value is assigned to the *catch* variable if one exists; otherwise, an error will be reported if no valid handler is present. The matching rules for exception handling are listed below:

1. *Basic Type:* Exceptions are caught if their type matches the `catch` type,

```
#include <cassert>
struct Base {};
struct Derived : Base{};

int main() {
  try {
    throw Derived();
  }
  catch(Base) {}
  catch(Derived) {assert(0);}
  return 0;
}
```

(a) Example of exception handling

```
   CATCH tag-Base->1, tag-Derived->2
   Derived tmp;
   THROW tag-Derived, tag-Base: tmp
   CATCH
   GOTO 3
1: Base
   GOTO 3
2: Derived
   ASSERT false
3: RETURN: 0
```

(b) GOTO program of exception handling

Figure 9: ESBMC verified an example with exception handling

ignoring qualifiers such as `const`, `volatile`, and `restrict`.

2. *Array and Pointer:* A pointer type in the `catch` block can catch exceptions of the corresponding array type.
3. *Function Pointer:* A `catch` block for a pointer to a function can catch exceptions of functions with the same return type.
4. *Base Class:* Exceptions derived unambiguously from the `catch` block's type are caught.
5. *Convertible Type:* Exceptions are caught if they can be converted to the type specified in the `catch` block through standard conversions or qualification adjustments.
6. *Void Pointer:* A `void*` in the `catch` block can catch any pointer type exception.
7. *Ellipsis:* Any exception can be caught using an ellipsis (`...`) in the `catch` block.
8. *Re-throw:* If no new exception is thrown, the last thrown exception should be re-thrown.

We have extended the symbolic engine to improve exception handling in ESBMC v7.6. As illustrated in Figure 9a, both exception handlers can catch the thrown exception. In ESBMC v7.6, the exception will be caught by *Base* in line 6, stopping the execution of subsequent exception handlers. Therefore, this sample code will not trigger the assertion, and the verification result will be successful.

As part of the exception handling mechanism, exception specifications clarify a function's exception behavior by defining which exceptions a function can throw. In ESBMC v2.1, we implemented Dynamic Exception Specification, which uses the *throw* keyword to declare a list of exception types that a function can throw, and the first line of Figure 10 illustrates the two types of exceptions that a function is allowed to throw: *int* and *double*. However, with updates to the C++ standard, the Dynamic Exception Specification was deprecated due to its limitations on flexibility. Consequently, we have supported Non-Dynamic Exception Specification in the new front-end. As shown in the third line of Figure 10, the *noexcept* keyword provides a modern way to declare a function's exception behavior. We used the THROW DECL instruction at the beginning of the function to check if any thrown exceptions match the exception specification. If the thrown exception violates the exception specification, it will result in an assertion property violation.

```
1 void func() throw(int, double);
2
3 void func() noexcept;
```

Figure 10: Example of exception specification

From Process Flow 5, it is clear that exception handling involves parsing the throw statement, try block, and catch block from the Clang AST. If the function includes an exception specification, the thrown exception is verified to conform to the specified constraints. When a throw statement is executed within a try block, a matching catch block is checked. If such a block can handle the exception, the execution jumps to it. Otherwise, an assertion violation is reported for the unhandled exception. Similarly, if a throw statement is executed outside any try block, an assertion violation is raised due to the absence of a handler. Furthermore, if the throw occurs during a function call and is not caught within that function, it will propagate to the calling function.

3.6. C++ Operational Model

ESBMC employs an abstract representation of the STL known as the C++ OMs, which are manually created and maintained. These models define function contracts, including pre- and post-conditions, for the STL functions and method calls they encompass, while also having side effects, such as

Process Flow 5 Exception Handling

Parse *throw* statement, *try* block and *catch* block from Clang AST
if Function has exception specification **then**
    Check that the thrown exception conforms to the specification
end if
if *Throw* statement is executed inside a *try* block **then**
    if A *catch* block exists and can catch that exception **then**
        Jump to the corresponding *catch* block
    else
        Assertion property violation: Failure to catch an exception
    end if
else if *Throw* statement is executed outside the *try* block **then**
    Assertion property violation: Failure to catch an exception
end if

exception propagation. Verifying all these simplifies verification. These OMs were developed based on the old front-end, which utilizes a CPROVER-based parse tree for its type checker. Therefore, the static checking capabilities of these OMs rely on maintenance. As the C++ standard updates, the code within these OMs has gradually become outdated.

With our new Clang-based front-end, static checking has become more compliant with C++ standards. This is due to Clang's following of language standards, advanced type deduction and checking mechanisms. Consequently, ESBMC v7.6 can now detect and report potential program issues during parsing. To adapt the OMs to the new front-end, we identified and addressed parsing errors in the OMs that required fixes, as shown in Table 1. Consequently, we updated the outdated code syntax, standardized variable names, and improved readability.

Category	**Operational models**
Containers	`vector`, `queue`, `deque`, `set`, `map`, `iterator`, `algorithm`, `stack`, `bitset`
Streams Input/Output	`istream`, `ios`, `ostream`, `sstream`, `fstream`, `streambuf`
Strings	`string`, `string_view`
Numeric	`numeric`, `valarry`
Language Support	`typeinfo`, `exception`
General	`memory`, `stdexcept`
Localization	`locale`

Table 1: Overview of the fixed C++ operational models

4. Experimental Evaluation

We used benchmarks from Monteiro et al. [13] to evaluate ESBMC v7.6, which were previously used to assess ESBMC v2.1 in the same study.

We used benchmarks to verify the core C++ language features. There are 532 test cases (TCs) in total over 6 benchmarks collections. The set of benchmarks *cpp* contains example programs from the book *C++ How to Program* [35]. The inheritance and polymorphism benchmarks are extracted from [13]. There are three benchmark collections for template specialization -- *cbmc-template* comes from the CBMC regressions [36]; *gcc-template-tests* were extracted from the GCC template test suite [31]; *template* is also from benchmarks used in [13]. The *cpp* set contains programs with mixed use of various C++ language features combined with inheritance, polymorphism, templates, and dynamic memory. Finally, we evaluated the 1001 TCs that depend on the OMs in each benchmark, and these test cases contain the most frequently used STL libraries.

4.1. Objectives and Setup

Our evaluation framework is based on *BenchExec* [37]. For each TC in the test suite, we check whether the verification verdict reported by each tool matches the expected outcome. A TC passes when the tool reports a verdict of "VERIFICATION SUCCESSFUL" on a program without any violation of properties or reports "VERIFICATION FAILED" on an unsafe program that violates a property. Such properties include arithmetic overflows, array out-of-bounds accesses, memory issues, or assertion failures. Our evaluation aims to answer the following experimental questions:

EQ1 (**soundness**): Can ESBMC v7.6 give more correct verification results and a higher pass rate than its previous versions?

EQ2 (**performance**): How long does ESBMC v7.6 take to verify C++ programs?

EQ3 (**completeness**): Does the tool implement the proposed improvements in complex template support outlined as future work by Monteiro et al. [13]?

The experiment was set up in Ubuntu 22.04 running on an 8-core Intel CPU and 16GB RAM, with a time limit of 900 seconds and a memory limit of 6GB. The dataset, scripts, and logs are publicly available on Zenodo [38].

The cumulative verification time represents the CPU time elapsed for each tool finishing all benchmarks.

4.2. Results

Table 2 shows our experimental results. With a higher pass rate than ESBMC v2.1 across all benchmarks and outperforming ESBMC v7.3 on 4 out of 6 benchmark sets, ESBMC v7.6 successfully achieved more correct results, confirming **EQ1**. As for ESBMC v2.1, the failed TCs in *cpp* are due to parsing or conversion errors, meaning the previous tool version is unable to properly typecheck the input programs, probably due to the weak parser, as described in Section 2. The failed TCs in *inheritance* and *polymorphism* set contain a common feature of dynamically casting a pointer of a child class with a base class containing virtual methods. ESBMC v2.1 could not handle this type of casting, giving conversion errors.

EQ1: ESBMC v7.6 has a higher pass rate across all benchmarks compared to v2.1 and v7.3, features a more powerful parser to support more C++ features.

ESBMC v2.1 has limited support for C++ templates, matching our expectations as reported by Monteiro et al. [13]. The failed TCs in the *cbmc-template* set are the results of ESBMC v2.1 not able to handle the default template type parameter or explicit template specialization combined with C++ *typedef* specifier. The low pass rate of ESBMC v2.1 on the *gcc-template-tests* set indicates that this version cannot verify test cases used by an industrial-strength compiler. **EQ3** is affirmed through the experiment, as none of these problems persist in ESBMC v7.6.

ESBMC v7.3 demonstrates better accuracy than v2.1 in features like inheritance and polymorphism, the failed TCs in the *cpp* and *template* sets are caused by outdated OMs and conversion errors. This indicates a lack of support for certain C++ syntax in the front-end and insufficient consideration of some edge cases during the conversion process. Regarding ESBMC v7.6, it addresses most of the parse and conversion errors in v7.3. However, some TCs fail because Clang generates different ASTs depending on the C++ standard, which the front-end does not yet fully support.

Only test cases verified correctly within the time limit are used for performance comparison. There were two timeouts for ESBMC v2.1 and none for v7.3 and v7.6. As illustrated in Figure 11, we conducted experiments

Benchmarks	ESBMC-v2.1	ESBMC-v7.3	ESBMC-v7.6
cpp	71%	61%	83%
inheritance	73%	93%	93%
polymorphism	80%	84%	85%
cbmc-template	92%	97%	97%
gcc-template-tests	39%	71%	78%
template	53%	65%	81%
Total verification Time	1090s	71s	545s

Table 2: Experimental results showing the pass rate for each set of benchmarks and accumulative verification time. This experiment uses ESBMC with Boolector SMT solver.

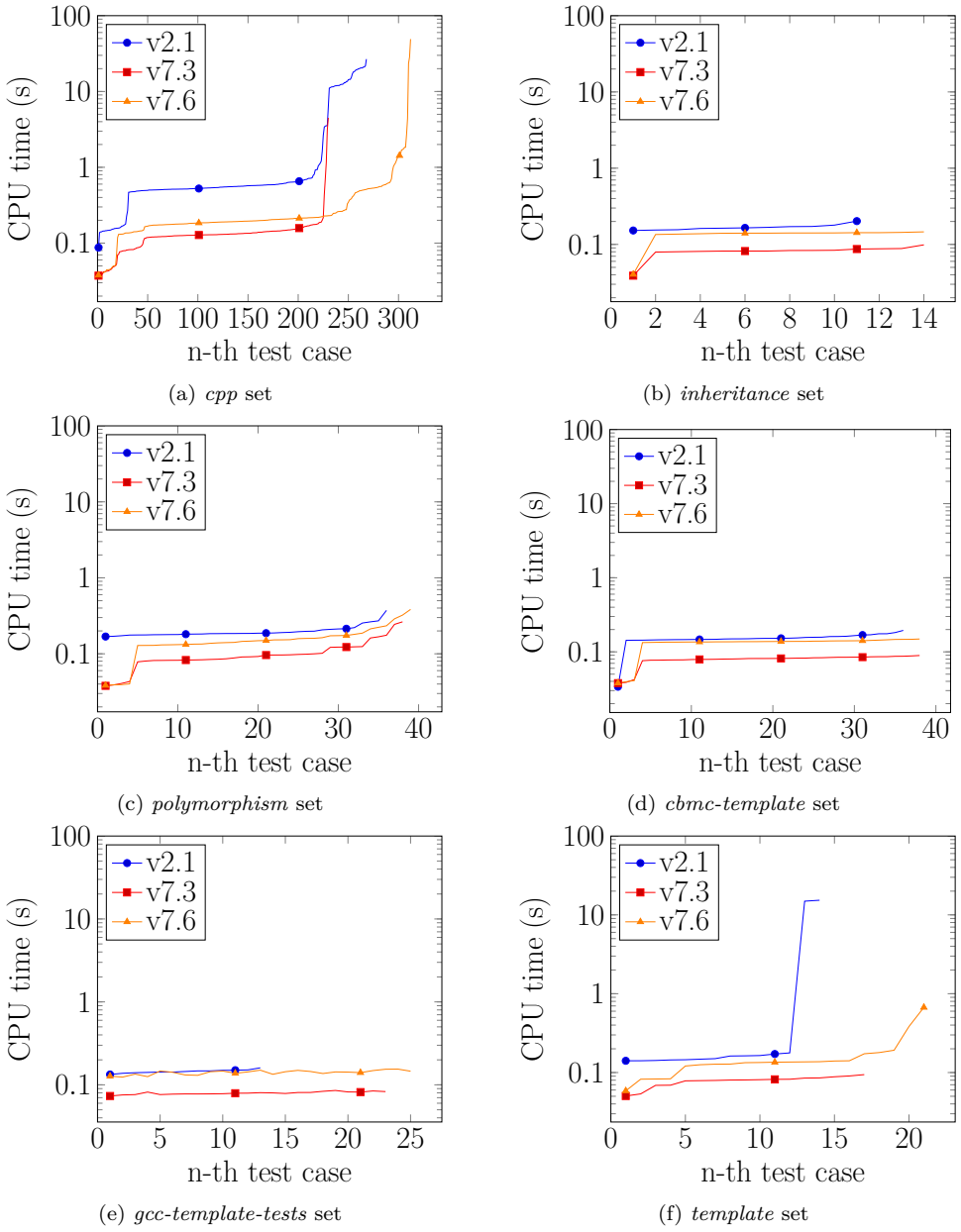
on each set of benchmarks. For simple benchmark sets like the *inheritance* and *polymorphism*, the runtime of the three ESBMC versions is comparable. However, for more complex benchmark sets, such as the *cpp* and *template*, the performance of v7.6 and v7.3, which use the Clang-based C++ front-end, shows improvements over v2.1. This is because the Clang-based front-end can efficiently parse complex C++ syntax and generate high-quality AST. The performance decrease from v7.6 to v7.3 results from the front-end's extended support for more advanced C++ features, which increases computational cost.

EQ2: ESBMC v7.6 is more efficient than v2.1, and although slightly slower than v7.3, it delivers better accuracy.

EQ3: ESBMC v7.6 handles complex template features more effectively and offers better extensibility.

Overall, we have enhanced the template support in ESBMC v7.6, which addresses a key aspect of the future work proposed by Monteiro et al. [13]. Compared to ESBMC v2.1, ESBMC v7.6 provides faster and more reliable performance. Although v7.6 takes slightly longer than v7.3, it delivers a notable improvement in the accuracy of results.

In addition to the pass rate and verification time in Table 2, we assessed each tool's memory usage. Table 3 shows each benchmark's cumulative maximum RSS (Resident Set Size) using each tool under evaluation. Our metrics collection approach is based on *BenchExec*'s efficient monitoring



(a) *cpp* set

(b) *inheritance* set

(c) *polymorphism* set

(d) *cbmc-template* set

(e) *gcc-template-tests* set

(f) *template* set

Figure 11: CPU time comparison for ESBMC versions on 6 benchmark sets (correct results only)

capabilities provided by the *control groups* memory subsystem [37]. Compared to ESBMC v2.1, ESBMC v7.6 has high pass rates and uses less memory in total. The increased memory usage in v2.1 for the *cpp* and *template* set is because of its inefficiency in verifying test cases involving templates due to its limitations in handling C++ templates and complex features effectively. Many TCs failed due to a CONVERSION ERROR in ESBMC v2.1's front-end and never even reached the solver in the back-end. Figure 12 shows that v2.1 uses less memory for simple TCs but consumes exponentially more for complex ones. ESBMC with a Clang-based front-end achieves stable and lower memory usage due to its efficient parsing and conversion processes.

Benchmarks	ESBMC-v2.1	ESBMC-v7.3	ESBMC-v7.6
cpp	219000 MB	13800 MB	26900 MB
inheritance	236 MB	490 MB	653 MB
polymorphism	722 MB	1480 MB	1960 MB
cbmc-template	643 MB	1260 MB	1680 MB
gcc-template-tests	457 MB	1030 MB	1390 MB
template	20800 MB	691 MB	1240 MB
Total memory	242000 MB	18700 MB	33800 MB

Table 3: Experimental results showing each benchmark's cumulative maximum RSS (Resident Set Size). This experiment uses ESBMC with the Boolector SMT solver.

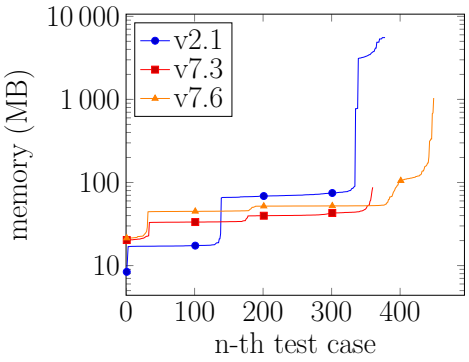


Figure 12: RSS comparison for ESBMC versions on all benchmarks (correct results only)

In ESBMC v2.1, we simulated the behavior of the C++ STL library using OMs and added safety properties. Since then, our C++ front-end has been completely rewritten based on Clang AST, and the back-end has undergone

significant development. Comparing v7.6 with v7.3, we have updated these outdated OMs and resolved their issues. We believe it is essential to re-evaluate v7.6 over the C++ STL library benchmarks [13] using these existing OMs.

As shown in Table 4, ESBMC v2.1 has generally high pass rates across most benchmarks, indicating strong support for OMs with security properties. In v7.3, the refactored front-end and outdated OMs resulted in poor pass rates due to the lack of support for several core language features. By comparison, the pass rates for most benchmarks have significantly improved with v7.6, with many returning to or exceeding the pass rates in v2.1. This indicates that the adaptation of the new front-end to the OMs has largely been resolved. Nevertheless, some benchmarks, such as *Multiset*, *Set*, and *Deque*, still lag behind the performance seen in v2.1. Most of the test cases failed due to parsing errors caused by initialization errors in the container OM. This indicates a need for further improvement in our OMs. Additionally, some errors arose from unsupported Clang AST nodes, and extending the front-end to support these AST nodes remains an ongoing development effort.

Benchmarks	ESBMC-v2.1	ESBMC-v7.3	ESBMC-v7.6
string	**99%**	0%	88%
stream	**89%**	33%	88%
algorithm	42%	0%	**80%**
deque	**95%**	0%	88%
list	53%	0%	**65%**
map	**83%**	0%	81%
multimap	89%	0%	**91%**
multiset	**74%**	0%	18%
priority-queue	**100%**	0%	87%
set	**83%**	0%	60%
stack	86%	0%	**86%**
vector	22%	0%	**89%**
try-catch	**88%**	0%	80%

Table 4: Pass rates of OM-dependent benchmarks for C++ STL libraries.

4.3. Performance Using Different SMT Solvers

ESBMC v7.6 supports multiple SMT solvers in the back-end, such as Z3 [39], Bitwuzla [40], Boolector [41], MathSAT [42], CVC4 [43], CVC5 [44] and Yices [45]. We also evaluated ESBMC v7.6 with various solvers over

the same set of benchmarks. Table 5 shows the total verification time and memory consumption for ESBMC v7.6 using different solvers.

Solvers	Time	Memory
Boolector	545s	33800 MB
CVC4	2230s	44100 MB
CVC5	1420s	42300 MB
MathSAT	2930s	50500 MB
Yices	1270s	40800 MB
Z3	1430s	34300 MB
Bitwuzla	549s	38700 MB

Table 5: Experimental results showing the total verification time and memory consumption for ESBMC v7.6 using different solvers.

Overall, ESBMC v7.6 with Boolector is the fastest configuration that consumes the minimum amount of memory to verify all benchmarks, while Bitwuzla performs similarly but consumes more memory. Among the other solvers, the memory consumption of ESBMC v7.6 with Z3 is close to the Boolector configuration.

4.4. Threats to Validity

While developing the new C++ front-end, we encountered challenges in determining the correct order of constructor and destructor calls for the most derived class when analyzing complex inheritance hierarchies in Clang AST, such as the diamond inheritance pattern. We documented it under an umbrella issue currently in our backlog [46] on ESBMC GitHub repository [47]. ESBMC v2.1 mimics the semantics of the APIs of C++ STL libraries using a set of OMs. The C++ front-end of ESBMC has been completely rewritten, and the back-end has also undergone significant development and evolution since v2.1 was published in [13]. Additionally, the number of these OMs is large, and for libraries without added safety properties, using the C++ standard library directly is the best solution. However, it is uncertain whether ESBMC's C++ front-end can fully support the standard library.

5. Conclusions and Future Work

We present a new Clang-based front-end that converts in-memory Clang AST to ESBMC's IR. In our evaluation of ESBMC v7.6, we compared it

to ESBMC v2.1 and v7.3, specifically focusing on benchmarks to cover core C++ language features. The results demonstrate significant progress with ESBMC v7.6, as it successfully handles real-world C++ programs, including those from the GCC test suite. Notably, it significantly reduces the number of conversion and parse errors compared to the previous version, showcasing improved performance over the benchmarks for core language features.

While ESBMC effectively mimics the semantics of APIs of the STL libraries using the OMs from ESBMC v2.1, we recognize the need for continuous improvement. As we endeavor to verify modern C++ programs, these OMs require regular review and updates to align with the C++ standard used in the input program. Accurate OMs are essential, as any approximation may lead to incorrect encoding and invalidate the verification results. With ESBMC v7.6, we improved the front-end, updated the OMs, and added support for more core C++ language features. Overall, while our Clang AST-based C++ front-end has not fully restored or improved performance across all benchmarks, the experimental results show substantial improvements compared to previous versions. This highlights the potential of the new front-end.

Furthermore, as part of the remaining future work from Monteiro et al. [13], our OMs have yet to support certain C++11 features, including new sequential and unordered associative containers, as well as multithreaded libraries, which remain areas for future development. Our previous success verifying a commercial C++ telecommunication application using ESBMC v2.1 has inspired further goals [48, 13]. With ESBMC v7.6 and beyond, we plan to verify the C++ interpreter in OpenJDK as part of the Soteria project [49].

6. Acknowledgements

The ESBMC development is currently funded by ARM, Intel, EPSRC grants EP/T026995/1, EP/V000497/1, EU H2020 ELEGANT 957286, and Soteria project awarded by the UK Research and Innovation for the Digital Security by Design (DSbD) Programme.

References

[1] K. Song, M. Ramalho, F. Brauße, R. Menezes, L. Cordeiro, ESBMC v7.3: Model Checking C++ Programs Using Clang AST, 2023, pp. 141–152. doi:10.1007/978-3-031-49342-3_9.

[2] P. J. Deitel, H. M. Deitel, C++ How to Program: Introducing the New C++14 Standard, 2016.

[3] M. Miller, Trends and Challenges in the Vulnerability Mitigation Landscape, USENIX Association (2019).

[4] J. Fan, Y. Li, S. Wang, T. N. Nguyen, A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries, in: Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 508–512.

[5] Common Vulnerabilities and Exposures database.
URL https://cve.mitre.org/

[6] Common Weakness Enumeration.
URL https://cwe.mitre.org/about/index.html

[7] S. Quadri, S. U. Farooq, Software Testing–Goals, Principles and Limitations, International Journal of Computer Applications 6 (9) (2010) 1.

[8] P. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, 2016.

[9] F. R. Monteiro, M. Garcia, L. C. Cordeiro, E. B. de Lima Filho, Bounded model checking of C++ programs based on the Qt cross-platform framework, Softw. Test. Verification Reliab. 27 (3) (2017). doi:10.1002/stvr.1632.
URL https://doi.org/10.1002/stvr.1632

[10] N. Chong, B. Cook, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, M. R. Tuttle, Code-level Model Checking in the Software Development Workflow, in: 2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), IEEE, 2020, pp. 11–20.

[11] E. Clarke, D. Kroening, F. Lerda, A Tool for Checking ANSI-C Programs, in: Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004,

Barcelona, Spain, March 29-April 2, 2004. Proceedings 10, Springer, 2004, pp. 168–176.

[12] Z. Baranová, J. Barnat, K. Kejstová, T. Kučera, H. Lauko, J. Mrázek, P. Ročkai, V. Štill, Model Checking of C and C++ With DIVINE 4, in: Automated Technology for Verification and Analysis: 15th International Symposium, ATVA 2017, Pune, India, October 3–6, 2017, Proceedings 15, Springer, 2017, pp. 201–207.

[13] F. R. Monteiro, M. R. Gadelha, L. C. Cordeiro, Model Checking C++ Programs, Software Testing, Verification and Reliability 32 (1) (2022) e1793. doi:https://doi.org/10.1002/stvr.1793.

[14] L. Cordeiro, B. Fischer, J. Marques-Silva, SMT-based Bounded Model Checking for Embedded ANSI-C Software, IEEE Transactions on Software Engineering 38 (4) (2011) 957–974.

[15] LLVM Clang.
URL https://clang.llvm.org/

[16] M. Ramalho, M. Freitas, F. Sousa, H. Marques, L. Cordeiro, B. Fischer, SMT-based Bounded Model Checking of C++ Programs, in: 2013 20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS), IEEE, 2013, pp. 147–156.

[17] C++20 Standard.
URL https://www.iso.org/standard/79358.html

[18] B. C. Lopes, R. Auler, Getting Started With LLVM Core Libraries, Packt Publishing Ltd, 2014.

[19] M. Pandey, S. Sarda, LLVM cookbook, Packt Publishing Ltd, 2015.

[20] N. M. Josuttis, The C++ Standard Library: A Tutorial and Reference (2012).

[21] G. Dos Reis, J. D. García, F. Logozzo, M. Fähndrich, S. Lahiri, Simple Contracts for C++(R1) (2015).

[22] S. Muchnick, Advanced Compiler Design Implementation, Morgan kaufmann, 1997.

[23] D. Kroening, J. Ouaknine, O. Strichman, T. Wahl, J. Worrell, Linear Completeness Thresholds for Bounded Model Checking, in: Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23, Springer, 2011, pp. 557–572.

[24] M. R. Gadelha, F. Monteiro, L. Cordeiro, D. Nicole, ESBMC v6. 0: Verifying C Programs Using k-Induction and Invariant Inference: (Competition Contribution), in: Tools and Algorithms for the Construction and Analysis of Systems: 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III 25, Springer, 2019, pp. 209–213.

[25] ESBMC L312-L359.
URL https://github.com/esbmc/esbmc/blob/master/src/cpp/cpp_typecheck_compound_type.cpp

[26] C++03 Standard.
URL https://www.iso.org/standard/38110.html

[27] L. C. Cordeiro, B. Fischer, Verifying Multi-Threaded Software Using SMT-based Context-Bounded Model Checking, in: R. N. Taylor, H. C. Gall, N. Medvidovic (Eds.), Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011, ACM, 2011, pp. 331–340. doi:10.1145/1985793.1985839.
URL https://doi.org/10.1145/1985793.1985839

[28] S. Prata, C++ Primer Plus, Pearson Education India, 2012.

[29] B. Stroustrup, The C++ Programming Language Fourth Edition (2013).

[30] J. Siek, W. Taha, A Semantic Analysis of C++ Templates, in: European Conference on Object-Oriented Programming, Springer, 2006, pp. 304–327.

[31] GCC test suite.
URL https://gcc.gnu.org/git/p=gcc.git;a=blob_plain;f=gcc/testsuite/g%2B%2B.dg/template/friend18.C;hb=649fc72d2

[32] GCC Bugzilla Bug 10158.
URL https://gcc.gnu.org/bugzilla/show_bug.cgi?id=10158

[33] CBMC 5.88.1.
URL https://github.com/diffblue/cbmc/releases/tag/cbmc-5.88.1

[34] Cppcheck.
URL https://cppcheck.sourceforge.io/

[35] P. Deitel, H. Deitel, C++ How to Program, Sixth Edition, Prentice Hall Press, USA, 2007.

[36] CBMC Regression Test Suite.
URL https://github.com/diffblue/cbmc/tree/develop/regression/cbmc-cpp

[37] D. Beyer, S. Löwe, P. Wendler, Reliable Benchmarking: Requirements and Solutions, Int. J. Softw. Tools Technol. Transf. 21 (1) (2019) 1–29. doi:10.1007/s10009-017-0469-y.
URL https://doi.org/10.1007/s10009-017-0469-y

[38] X. Li, ESBMC v7.6 evaluation (2025).
URL https://zenodo.org/records/14824495

[39] L. d. Moura, N. Bjørner, Z3: An Efficient SMT Solver, in: International conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2008, pp. 337–340.

[40] A. Niemetz, M. Preiner, Bitwuzla, in: C. Enea, A. Lal (Eds.), Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II, Vol. 13965 of Lecture Notes in Computer Science, Springer, 2023, pp. 3–17.

[41] R. Brummayer, A. Biere, Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2009, pp. 174–177.

[42] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani, The MathSat 4 SMT Solver, in: International Conference on Computer Aided Verification, Springer, 2008, pp. 299–303.

[43] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, C. Tinelli, Cvc4, in: International Conference on Computer Aided Verification, Springer, 2011, pp. 171–177.

[44] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, et al., cvc5: A Versatile and Industrial-Strength SMT Solver, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2022, pp. 415–442.

[45] B. Dutertre, Yices 2.2, in: International Conference on Computer Aided Verification, Springer, 2014, pp. 737–744.

[46] ESBMC Cpp Support Feature Coverage and Backlog.
URL https://github.com/esbmc/esbmc/wiki/ESBMC-Cpp-Support

[47] Github, ESBMC Issue 940: Umbrella Issue For the Order of Ctors/Dtors.
URL https://github.com/esbmc/esbmc/issues/940

[48] F. R. M. Sousa, L. C. Cordeiro, E. B. de Lima Filho, Bounded Model Checking of C++ Programs Based on the Qt Framework, in: IEEE 4th Global Conference on Consumer Electronics, GCCE 2015, Osaka, Japan, 27-30 October 2015, IEEE, 2015, pp. 179–180. doi:10.1109/GCCE.2015.7398699.
URL https://doi.org/10.1109/GCCE.2015.7398699

[49] UKRI, Sotereia Project.
URL https://soteriaresearch.org/