ESBMC v7.3: Model Checking C++ Programs using Clang AST

Kunjian Song[1], Mikhail R. Gadelha[2], Franz Brauße[1], Rafael S. Menezes[1], and Lucas C. Cordeiro[1]

[1] The University of Manchester, UK
{kunjian.song, rafael.menezes}@postgrad.manchester.ac.uk,
{franz.brausse, lucas.cordeiro}@manchester.ac.uk
[2] Igalia, A Coruña, Spain
mikhail@igalia.com

Abstract. This paper introduces ESBMC v7.3, the latest Efficient SMT-Based Context-Bounded Model Checker version, which now incorporates a new clang-based C++ front-end. While the previous CPROVER-based front-end served well for handling C++03 programs, it encountered challenges keeping up with the evolving C++ language. As new language and library features were added in each C++ version, the limitations of the old front-end became apparent, leading to difficult-to-maintain code. Consequently, modern C++ programs were challenging to verify. To overcome this obstacle, we redeveloped the front-end, opting for a more robust approach using clang. The new front-end efficiently traverses the Abstract Syntax Tree (AST) in-memory using clang APIs and transforms each AST node into ESBMC's Intermediate Representation. Through extensive experimentation, our results demonstrate that ESBMC v7.3 with the new front-end significantly reduces parse and conversion errors, enabling successful verification of a wide range of C++ programs, thereby outperforming previous ESBMC versions.

Keywords: Formal Methods · Model Checking · Software Verification

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 [1,2]. 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]. Unlike testing, formal verification techniques

can be used more systematically to 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 [1]. The existing state-of-the-art verification tools for C++ programs only have limited feature support [11]. For ESBMC, Ramalho et al. [12] and Monteiro et al. [11] initiated the support for C++ program verification. Since then, ESBMC has undergone heavy development.

This research presents a significant improvement to ESBMC's C++ verification capabilities by introducing a new clang-based frontend. Particularly, the original contributions of this work are as follows:

- **Complete Redesign**: ESBMC's C++ frontend has undergone a complete overhaul and now relies on clang [13]. By leveraging Clang's parsing and semantic analysis capabilities [14,15], we check the input program's Abstract Syntax Tree (AST) using a production-quality compiler. This eliminates the need for 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). This improvement allows ESBMC to handle C++ growth and its variants like CUDA [16].
- **Simplified Type Checking for Templates:** The new clang-based frontend greatly simplifies type checking for templates, streamlining ESBMC's ability to adapt to C++ advancements. Furthermore, this enhancement facilitates the incorporation of C++ variants like CUDA.

By introducing these advancements, our work significantly enhances ESBMC's C++ verification capabilities, paving the way for more robust and efficient verification of C++ programs and their variants.

2 Background

ESBMC's verification for C++03 programs reaches its maturity in version v2.1, presented by Monteiro et al. [11]. 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 (OM) 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 [17]. Combining these approaches, ESBMC v2.1 outperformed other state-of-the-art tools evaluated over a large set of benchmarks, comprising 1513 test cases [11]. Nonetheless, ESBMC v2.1 employs a Flex and Bison-based frontend from CBMC [18], which leads to hard-to-main code and can hardly evolve to support modern C++11 features.

Limitations of the old C++ frontend The version of ESBMC in Monteiro et al. [11] uses an outdated CPROVER-based frontend [18] with the following limitations.

1. For the type-checking phase, ESBMC could not provide meaningful warnings or error messages.
2. It is 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 needs to be manually updated to cover the essential C++ semantic rules [19], which leads to hard-to-maintain code to keep up with the C++ evolution.
4. The old front-end contains excessive data structures and procedures auxiliary to scope resolution and function type checking.
5. The type checker [19] of the old frontend only works with a CPROVER-based parse tree and supports up to C++03 standard [20]. We find adapting it to the new C++ language and library features difficult.
6. The old front-end uses 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 on the templates.

These limitations combine to a point where the old front-end is too laborious to maintain and extend for formal verification of modern C++ programs. We propose the clang-based approach to convert an input C++ program to ESBMC's IR to overcome these limitations.

3 Model Checking C++ Programs using Clang AST

Figure 1 illustrates ESBMC's verification pipeline for C++ programs. The new clang-cpp frontend type-checks and converts the input C++ program (along with the corresponding OMs) into the GOTO program representation [21,22]. 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 the bound k or a counterexample in case of property violation.

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, to the best of our knowledge, does not include information about virtual tables or virtual pointers of a class, it nonetheless provides users with enough information to enable them to create their *vtables* and *vptrs*. In the new clang-based C++ frontend, we reimplemented the *vtable* and *vptr* construction mechanism following a similar approach from ESBMC v2.1, but with significant

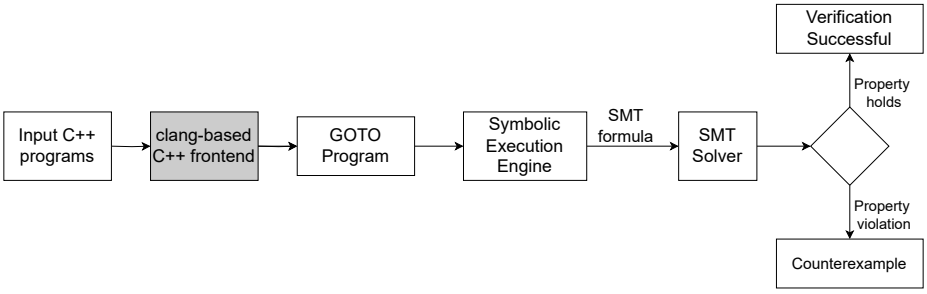

Fig. 1: ESBMC architecture for C++ verification. The grey block represents the new clang-based C++ front-end integrated into ESBMC v7.3.

```
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 int main(){
11   Bird *p = new Penguin();
12   assert(p->doit() == 42);
13   delete p;
14   return 0;
15 }
```

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

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 Bird and Penguin classes. 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 a *vtable* with a thunk to the overriding function. The thunk redirects the call to the corresponding overriding function. In the case of multiple inheritances, the child class would have multiple *vtprs* "inherited" from multiple base classes. The new front-end can also manage a virtual inheritance, such as the diamond problem, which avoids duplicating *vptrs*, referring to the same virtual table in an inheritance hierarchy. Line 2-4 in Figure 4a illustrates the dynamic dispatch is achieved using the *vptr* calling the thunk, which in turn calls the desired overriding function in Figure 4b Line 9-11. Note that the *override* specifier is a C++11 extension that the old front-end could not support.

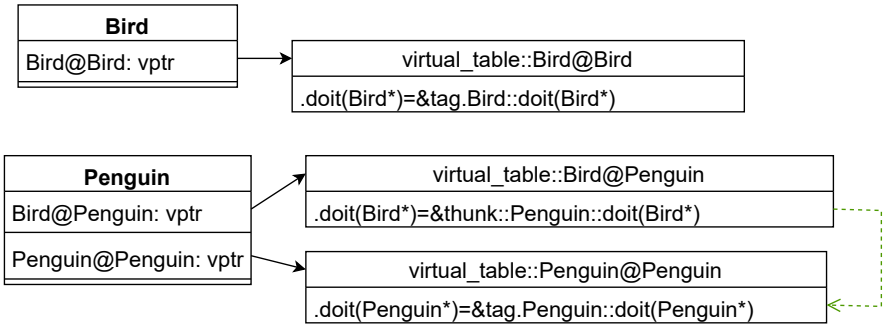

Fig. 3: Object models for Bird and Penguin classes

```
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.

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

3.2 Template

Template is a key feature in C++, allowing type to be passed as a parameter. The template allows STL containers and generic algorithms to work with different C++ data types [23,24]. The old front-end in ESBMC v2.1 implements the template specialization based on Siek et al. [25,11]. 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 [26], which was added for Bug 10158 on GCC Bugzilla [27]. ESBMC v7.3 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 [28], which aborted during type-checking, and cppcheck v2.11.1 [29], which did not give any verification verdict.

```cpp
#include <cassert>
template <int N> struct X
{
  template <int M>
  friend int foo(X const &)
  {
    return N * 10000 + M;
  }
};
X<1234> bring;

int main() {
  assert(
   foo<5678> (bring)
    !=12345678);
}
```

(a) Example of C++ class template

```
Violated property:
  file tmp2.cpp
   line 13 column 3
    function main
  assertion
   foo<5678>(bring)!=12345678
  return_value!=12345678

VERIFICATION FAILED
```

(b) Verdict for the template example

Fig. 5: ESBMC verified the *Friend18* example from the GCC test suite. [26]

4 Experimental Evaluation

We used some benchmarks from Monteiro et al. [11] to evaluate ESBMC v7.3. These benchmarks were used to assess ESBMC v2.1. Still, we excluded the other two verifiers (LLBMC [30] and DIVINE [31]) for the following reasons: (1) they have been already evaluated by Monteiro et al. [11] and ESBMC v2.1 was found outperforming them; (2) to our best knowledge and effort, we were unable to get a working version of them. The last version of LLBMC was released in 2013, and its download link is currently broken. The last commit to DIVINE's mirror repository on GitHub dated back to Mar 2021, and DIVINE failed to build.

We did not evaluate the test cases (TCs) that depend on the operational models (OMs) in each benchmark. We only ran the TCs for core C++ language features because the OMs for the new clang-based C++ front-end are still under development, e.g., exception handling support. Otherwise, running test cases for sure to fail would be pointless due to a feature still being developed. Hence each benchmark is a subset of the original benchmark, which only comprises TCs for verifying core C++ language features. There are 399 benchmarks in total over 6 sub-benchmarks. The *cpp-sub* contains example programs from the book *C++ How to Program* [32]. The inheritance and polymorphism sub-benchmarks are extracted from [11]. There are three sub-benchmarks for template specialization - *cbmc-sub* comes from the CBMC regressions [33]; *gcc-template-tests-sub* were extracted from the GCC template test suite [26]; *template-sub* is also from benchmarks used in [11]. *cpp-sub* contains programs with mixed use of various C++ language features combined with inheritance, polymorphism, and templates.

4.1 Objectives and Setup

Our evaluation framework is based on Python's *unittest* [34]. For each TC in the test suite, we check whether the verification verdict reported by each

tool matches the expected outcome. 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, memory issues, or assertion failures. Our evaluation aims to answer the following experimental questions:

EQ1 : (**soundness**) Can ESBMC give more correct verification results and a higher pass rate than its previous versions?
EQ2 : (**performance**) How long does ESBMC v7.3 take to verify C++ programs?
EQ3 : (**completeness**) Does the tool complete the future work specified by Monteiro et al. [11]?

The experiment was set up in Ubuntu 20.04 with 32GB RAM on an 8-core Intel CPU. The dataset, scripts, and logs are publicly available in Zenodo [35]. The accumulative verification time represents the CPU time elapsed for each tool finishing all sub-benchmarks.

4.2 Results

Table 1 shows our experimental results. With a higher pass rate than ESBMC v2.1 over 5 out of 6 sub-benchmarks, ESBMC v7.3 successfully verified all benchmarks and passed all test cases, confirming **EQ1**. As for ESBMC v2.1, the failed TCs in *cpp-sub* are due to parsing or conversion errors, meaning the previous tool version is unable to properly type-check the input programs, probably due to the weak parser, as described in Section 2. The failed TCs in *inheritance and polymorphism-sub* 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.

ESBMC v2.1 has limited support for C++ templates, matching our expectations as reported by Monteiro et al. [11]. The failed test cases in *cbmc-template-sub* 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 *gcc-template-tests-sub* indicates that the old version cannot verify test cases used by an industrial compiler. **EQ3** is affirmed through the experiment, as none of these problems persist in ESBMC v7.3. Since one of the test cases in *cpp-sub* timed out against ESBMC v2.1 after 900 seconds, the actual verification time has been rectified to 149s; otherwise, the cumulative verification time would be 1049s. As for the performance **EQ2**, ESBMC v7.3 could verify all sub-benchmarks in 128s, faster than its previous version, which affirms **EQ2**.

Overall, we have enhanced the template support in ESBMC v7.3, which completed the future work by Monteiro et al. [11]. In comparison to its previous version, ESBMC v7.3 can provide more accurate results faster.

4.3 Threats to Validity

While developing the new C++ frontend, we found that the clang AST does not fully describe the correct order of constructors or destructors to be called in the most derived class in a complex hierarchical inheritance graph, e.g., crossed

Table 1: Experimental results showing the pass rate for each sub-benchmark and accumulative verification time.

Sub-Benchmarks	**ESBMC-v2.1 pass rate**	**ESBMC-v7.3 pass rate**
cpp-sub	91%	100%
inheritance-sub	79%	100%
polymorphism-sub	87%	100%
cbmc-template-sub	92%	100%
gcc-template-tests-sub	39%	100%
template-sub	100%	100%
Total verification Time	149.94s	128.796s

diamond hierarchy. We documented it under an umbrella issue, which is currently in our backlog [36] on ESBMC GitHub repository [37]. We might need to use an additional data structure to keep track of the most derived class and implement an algorithm to recursively describe the correct order of base initialization or destruction in the class inheritance graph, which remains an open challenge.

5 Conclusion 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 v7.3, we compared it to ESBMC v2.1, specifically focusing on a subset of benchmarks to cover core C++ language features. The results demonstrate significant progress with ESBMC v7.3, as it successfully parses 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 sub-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. To further enhance our front-end coverage and reduce the number of OMs we maintain, our future work will focus on handling more C++ libraries.

Additionally, we aim to integrate various checkers, such as cppcheck [29], into our testing framework to facilitate future evaluations. Our previous success verifying a commercial C++ telecommunication application using ESBMC v2.1 has inspired further goals [38,11]. With ESBMC v7.3 and beyond, we plan to verify the C++ interpreter in OpenJDK as part of the Soteria project [39] and contribute to benchmarks for the International Competition on Software Verification (SV-COMP) [40].

References

1. Deitel, P.J., Deitel, H.M.: C++ how to program: Introducing the new C++14 standard. Prentice Hall (2016)
2. Cordeiro, L.C., de Lima Filho, E.B., de Bessa, I.V.: Survey on automated symbolic verification and its application for synthesising cyber-physical systems. IET Cyper-Phys. Syst.: Theory & Appl. **5**(1), 1–24 (2020). https://doi.org/10.1049/iet-cps.2018.5006, `https://doi.org/10.1049/iet-cps.2018.5006`
3. Miller, M.: Trends and challenges in the vulnerability mitigation landscape. USENIX Association (2019)
4. Fan, J., Li, Y., Wang, S., Nguyen, T.N.: A C/C++ code vulnerability dataset with code changes and cve summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories. pp. 508–512 (2020)
5. Common Vulnerabilities and Exposures database, `https://cve.mitre.org/`
6. Common Weakness Enumeration, `https://cwe.mitre.org/about/index.html`
7. Quadri, S., Farooq, S.U.: Software testing–goals, principles and limitations. International Journal of Computer Applications **6**(9), 1 (2010)
8. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2016)
9. Monteiro, F.R., Garcia, M., Cordeiro, L.C., de Lima Filho, E.B.: Bounded model checking of C++ programs based on the Qt cross-platform framework. Softw. Test. Verification Reliab. **27**(3) (2017). https://doi.org/10.1002/stvr.1632, `https://doi.org/10.1002/stvr.1632`
10. Chong, N., Cook, B., Kallas, K., Khazem, K., Monteiro, F.R., Schwartz-Narbonne, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: 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). pp. 11–20. IEEE (2020)
11. Monteiro, F.R., Gadelha, M.R., Cordeiro, L.C.: Model checking C++ programs. Software Testing, Verification and Reliability **32**(1), e1793 (2022)
12. Ramalho, M., Freitas, M., Sousa, F., Marques, H., Cordeiro, L., Fischer, B.: Smt-based bounded model checking of c++ programs. In: 2013 20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS). pp. 147–156. IEEE (2013)
13. LLVM clang, `https://clang.llvm.org/`
14. Lopes, B.C., Auler, R.: Getting started with LLVM core libraries. Packt Publishing Ltd (2014)
15. Pandey, M., Sarda, S.: LLVM cookbook. Packt Publishing Ltd (2015)
16. Pereira, P.A., Albuquerque, H.F., da Silva, I., Marques, H., Monteiro, F.R., Ferreira, R., Cordeiro, L.C.: Smt-based context-bounded model checking for CUDA programs. Concurr. Comput. Pract. Exp. **29**(22) (2017). https://doi.org/10.1002/cpe.3934, `https://doi.org/10.1002/cpe.3934`
17. Dos Reis, G., García, J.D., Logozzo, F., Fähndrich, M., Lahiri, S.: Simple contracts for C++(R1) (2015)
18. Clarke, E., Kroening, D., Lerda, F.: 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. pp. 168–176. Springer (2004)
19. ESBMC L312-L359, `https://github.com/esbmc/esbmc/blob/master/src/cpp/cpp_typecheck_compound_type.cpp`
20. C++03 standard, `https://www.iso.org/standard/38110.html`
21. Cordeiro, L., Fischer, B., Marques-Silva, J.: Smt-based bounded model checking for embedded ANSI-C software. IEEE Transactions on Software Engineering **38**(4), 957–974 (2011)

22. Cordeiro, L.C., Fischer, B.: Verifying multi-threaded software using smt-based context-bounded model checking. In: Taylor, R.N., Gall, H.C., Medvidovic, N. (eds.) Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011. pp. 331–340. ACM (2011). https://doi.org/10.1145/1985793.1985839, https://doi.org/10.1145/1985793.1985839
23. Prata, S.: C++ primer plus. Pearson Education India (2012)
24. Stroustrup, B.: The C++ programming language fourth edition (2013)
25. Siek, J., Taha, W.: A semantic analysis of C++ templates. In: European Conference on Object-Oriented Programming. pp. 304–327. Springer (2006)
26. GCC test suite, https://gcc.gnu.org/git/?p=gcc.git;a=blob_plain;f=gcc/testsuite/g%2B%2B.dg/template/friend18.C;hb=649fc72d2
27. GCC bugzilla bug 10158, https://gcc.gnu.org/bugzilla/show_bug.cgi?id=10158
28. CBMC 5.88.1, https://github.com/diffblue/cbmc/releases/tag/cbmc-5.88.1
29. cppcheck, https://cppcheck.sourceforge.io/
30. LLBMC, https://llbmc.org/downloads.html
31. DIVINE mirror repository, https://github.com/paradise-fi/divine
32. Deitel, P.: C++ How To Program (6th edn.). Prentice Hall Press (2007)
33. CBMC regression test suite, https://github.com/diffblue/cbmc/tree/develop/regression/cbmc-cpp
34. Python unittest, https://docs.python.org/3/library/unittest.html
35. ESBMC v7.3 evaluation archive on Zenodo, https://zenodo.org/record/8233714
36. ESBMC cpp support feature coverage and backlog, https://github.com/esbmc/esbmc/wiki/ESBMC-Cpp-Support
37. Github: Esbmc issue 940: Umbrella issue for the order of ctors/dtors, https://github.com/esbmc/esbmc/issues/940
38. Sousa, F.R.M., Cordeiro, L.C., de Lima Filho, E.B.: 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. pp. 179–180. IEEE (2015). https://doi.org/10.1109/GCCE.2015.7398699, https://doi.org/10.1109/GCCE.2015.7398699
39. UKRI: Sotereia project, https://soteriaresearch.org/
40. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: 29th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 13994, pp. 495–522. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29, https://doi.org/10.1007/978-3-031-30820-8_29
41. Moura, L.d., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
42. Niemetz, A., Preiner, M.: Bitwuzla at the smt-comp 2020. arXiv preprint arXiv:2006.01621 (2020)
43. Brummayer, R., Biere, A.: Boolector: An efficient smt solver for bit-vectors and arrays. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 174–177. Springer (2009)
44. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The mathsat 4 smt solver. In: International Conference on Computer Aided Verification. pp. 299–303. Springer (2008)
45. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: International Conference on Computer Aided Verification. pp. 171–177. Springer (2011)
46. Dutertre, B.: Yices 2.2. In: International Conference on Computer Aided Verification. pp. 737–744. Springer (2014)

A Memory Consumption

In addition to the pass rate and verification time in Table 1, we also assessed each tool's memory usage. Table 2 shows the cumulative maximum RSS (Resident Set Size) for each benchmark using each tool under evaluation. Our metrics collection approach is based on Python's *resource* module [1], *subprocess* module [2] and *unit test framework* [34].

Memory metrics collection approach The unit test framework encapsulates a benchmark in a test suite that launches a sub-process for each test case and waits for it to finish. We obtain the maximum RSS for each test case sub-process that has been terminated. In each row of Table 2, the cumulative maximum RSS for a benchmark is calculated by summing the maximum RSS for each sub-process. The final row, *Total memory*, totals the amount of memory used by each tool to perform each benchmark.

Table 2: Experimental results showing the cumulative maximum RSS (Resident Set Size) for each sub-benchmarks.

Sub-Benchmarks	**ESBMC-v2.1**	**ESBMC-v7.3**
cpp-sub	31477 MB	19385 MB
inheritance-sub	231 MB	845 MB
polymorphism-sub	722 MB	2373 MB
cbmc-template-sub	650 MB	2295 MB
gcc-template-tests-sub	395 MB	1387 MB
template-sub	207 MB	727 MB
Total memory	33682 MB	27012 MB

Compared to ESBMC v2.1, ESBMC v7.3 can verify more test cases and uses less memory. The lower memory usage of v2.1 than v7.3 is due to lower pass rates for the benchmarks, mainly because of v2.1's inadequacy to handle C++ templates. Many TCs failed due to CONVERSION ERROR in ESBMC's front-end and never even reached the solver in ESBMC's backend. As a result, no verification effort was made for those TCs and hence less memory was used.

B Performance Using Different SMT Solvers

ESBMC supports multiple SMT solvers in the back-end, such as Z3 [41], Bitwuzla [42], Boolector [43], MathSAT [44], CVC4 [45], and Yices [46]. We also evaluated ESBMC v7.3 with various solvers over the same set of benchmarks. Table 3 shows the pass rates and total verification time for ESBMC v7.3 using different solvers, and Table 4 shows the memory consumption for the same

[1] `https://docs.python.org/3/library/resource.html`
[2] `https://docs.python.org/3/library/subprocess.html`

experimental set-up using the same metrics collection approach explained in Appendix A.

Overall, ESBMC v7.3 with Boolector is the fastest configuration that also consumes the minimum amount of memory to verify all benchmarks. Among the other solvers, the memory consumption of ESBMC v7.3 with Bitwuzla comes near the Boolector configuration. This is probably because Bitwuzla is an extended forked Boolector [42]. We also found that MathSAT tends to use more memory for timed-out test cases, as ESBMC v7.3 with MathSAT failed only one TC in the *cppsub* benchmark due to timeout but uses more memory than the other configurations.

Table 3: Experimental results showing the pass rate and total verification time for ESBMC using different solvers.

Sub-Benchmarks	**Boolector**	**CVC4**	**MathSAT**	**Yices**	**Z3**	**Bitwuzla**
cpp-sub	100%	99%	99%	100%	100%	100%
inheritance-sub	100%	93%	100%	100%	100%	100%
polymorphism-sub	100%	100%	100%	100%	100%	100%
cbmc-template-sub	100%	97%	100%	100%	100%	100%
gcc-template-tests-sub	100%	96%	100%	100%	100%	100%
template-sub	100%	92%	100%	100%	100%	100%
Total verification Time	128.796s	637.988s	131.934s	182.327s	162.848s	152.442

Table 4: Experimental results showing the memory usage for ESBMC using different solvers.

Sub-Benchmarks	**Boolector**	**CVC4**	**MathSAT**	**Yices**	**Z3**	**Bitwuzla**
cpp-sub	19385 MB	63757 MB	153326 MB	27983 MB	35758 MB	19455 MB
inheritance-sub	845 MB	950 MB	940 MB	847 MB	946 MB	855 MB
polymorphism-sub	2373 MB	2657 MB	2632 MB	2320 MB	2596 MB	2387 MB
cbmc-template-sub	2295 MB	2558 MB	2449 MB	2308 MB	2457 MB	2299 MB
gcc-template-tests-sub	1387 MB	1559 MB	1480 MB	1401 MB	1497 MB	1395 MB
template-sub	727 MB	800 MB	781 MB	730 MB	774 MB	733 MB
Total memory	27012 MB	72281 MB	161608 MB	35589 MB	44028 MB	27124 MB

C Planning for Future work

ESBMC v2.1 mimics the semantics of the APIs of C++ STL libraries using a set of operational models (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 [11], therefore it is questionable whether those OMs still work. We believe that it is essential to evaluate both ESBMC v2.1 and v7.3 with the existing OMs over the C++ library benchmarks from [11]. Table 5 provides a summary of the pass rates.

As shown in Table 5, the OMs of ESBMC v2.1 give a fairly good pass rate of 80% and more in most of the benchmarks, except for STL *algorithm*, *list*, *multiset* and *vector*, where the pass rate is below 50%. In ESBMC, there are a total of 63 C++ OMs representing the most frequently used STL libraries. 41 out of 63 OMs have been enabled with the new clang C++ front-end. The remaining OMs are still being fixed. As we continue to enable more OMs, the progress is tracked and publicly available for viewing on ESBMC's wiki page *OM Workload Estimate and Tracking* [3].

Table 5: Pass rates of OM-dependent benchmarks for C++ STL libraries.

Benchmarks	**ESBMC-v2.1 pass rate**	**ESBMC post-v7.3 pass rate**
string	99%	0%
stream	89%	33%
algorithm	42%	0%
deque	95%	0%
list	53%	0%
map	83%	0%
multimap	89%	0%
multiset	74%	0%
priority_queue	100%	0%
set	83%	0%
stack	86%	0%
vector	22%	0%
try_catch	88%	0%

[3] `https://github.com/esbmc/esbmc/wiki/OM-Workload-Estimate-and-Tracking`