FORMAL VERIFICATION TO ENSURING THE MEMORY SAFETY OF C++ PROGRAMS

A DISSERTATION
SUBMITTED TO THE POSTGRADUATE PROGRAM IN INFORMATICS
OF FEDERAL UNIVERSITY OF AMAZONAS
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Felipe R. Monteiro
April 2020

© Copyright by Felipe R. Monteiro 2020
All Rights Reserved

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

S725v

Sousa, Felipe Rodrigues Monteiro
Formal Verification to Ensuring the Memory Safety of C++ Programs / Felipe Rodrigues Monteiro Sousa . 2020
71 f.: il. color; 31 cm.

Orientador: Lucas Carvalho Cordeiro
Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

1. Software Verification. 2. Model Checking. 3. c++. 4. Memory Safety. 5. Engenharia de Software. I. Cordeiro, Lucas Carvalho. II. Universidade Federal do Amazonas III. Título

PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
INSTITUTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

FOLHA DE APROVAÇÃO

"Formal Verification to Ensuring the Memory Safety of C++ Programs"

FELIPE RODRIGUES MONTEIRO SOUSA

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Lucas Carvalho Cordeiro - PRESIDENTE

Prof. Raimundo da Silva Barreto - MEMBRO INTERNO

Prof. Herbert Oliveira Rocha - MEMBRO EXTERNO

Manaus, 17 de Janeiro de 2020

Abstract

In the last three decades, memory safety issues in low-level programming languages such as C or C++ have been one of the significant sources of security vulnerabilities; however, there exist only a few attempts with limited success to cope with the complexity of C++ program verification. This work describes and evaluates a novel verification approach based on bounded model checking (BMC) and satisfiability modulo theories (SMT) to verify C++ programs formally. This verification approach analyzes bounded C++ programs by encoding various sophisticated features that the C++ programming language offers into SMT, such as templates, sequential and associative containers, inheritance, polymorphism, and exception handling. We formalize these sophisticated 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 this verification approach on top of the Efficient SMT-Based Context-Bounded Model Checker (ESBMC). We compare ESBMC to LLBMC and DIVINE, which are state-of-the-art verifiers to check C++ programs directly from LLVM bitcode. The experimental evaluation contains a set of over $1,500$ benchmarks from several sources (e.g., Deitel & Deitel, NEC Corporation, and GCC test suite), which covers several C++ features. Experimental results show that ESBMC can handle a wide range of C++ programs, presenting a higher number of correct verification results, and at the same time, it reduces the verification time if compared to LLBMC and DIVINE tools.

Keywords: Software Verification, Model Checking, C++, Memory Safety.

Acknowledgments

I would like to thank my supervisor, Dr. Cordeiro, for his support, guidance, encouragement, and friendship during the last seven years. His support helped me to become a better professional and a better researcher throughout my academic life. Thanks for allowing me to be part of your research group. Many thanks to my colleagues from the Federal University of Amazonas for helping me with many valuable insights in my research, and for always encouraged me to not give up during the hard times. In fact, I would also like to thank all my friends for always being there for me. Thanks for the fun moments, and to remind me to put the best of me in every situation. Most importantly, I would especially like to thank my family for the love, support, and constant encouragement I have gotten over the years. In particular, I would like to thank my parents and my grandma for their help in my education and for having always believed in me. I dedicate this dissertation to them. I also dedicate this work to my Gabriel, for giving me all the love I need to keep ongoing.

Publications

i. **Felipe R. Monteiro**, Erickson H. da S. Alves, Isabela S. Silva, Hussama I. Ismail, Lucas C. Cordeiro, and Eddie B. de Lima Filho. 2018. *ESBMC-GPU – A Context-Bounded Model Checking Tool to Verify CUDA Programs*. Sci. Comput. Program. 152, C, 63-69, 2018. DOI: `https://doi.org/10.1016/j.scico.2017.09.005`.

ii. **Felipe R. Monteiro**, Mario A. P. Garcia, Lucas C. Cordeiro, and Eddie B. de Lima Filho. *Bounded Model Checking of C++ Programs based on the Qt Cross-Platform Framework (Journal-First Abstract)*. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18). ACM, New York, NY, USA, 954–954, 2018. DOI: `https://doi.org/10.1145/3238147.3241981`.

iii. Mikhail R. Gadelha, **Felipe R. Monteiro**, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. *ESBMC 5.0: An Industrial-Strength C Model Checker*. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18). ACM, New York, NY, USA, 888-891, 2018. DOI: `https://doi.org/10.1145/3238147.3240481`.

iv. Mikhail R. Gadelha, **Felipe R. Monteiro**, Lucas C. Cordeiro, and Denis A. Nicole. *Towards Counterexample-Guided k-Induction for Fast Bug Detection*. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18). ACM, New York, NY, USA, 765-769, 2018. DOI: `https://doi.org/10.1145/3236024.3264840`.

v. **Felipe R. Monteiro**, Mikhail R. Gadelha, and Lucas C. Cordeiro. *Continuous Formal Verification at Scale*. In 10th Workshop on Tools for Automatic Program Analysis (TAPAS'19). Online proceedings, Porto, Portugal, 2019.

vi. Mikhail R. Gadelha, **Felipe R. Monteiro**, Lucas C. Cordeiro, and Denis A. Nicole. *Scalable and Precise Verification based on k-Induction and Floating-Point Theory*. In Proceedings of the 23rd International Conference on Fundamental Approaches to Software Engineering (FASE'20). Springer International Publishing, 2020.

vii. Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, **Felipe R. Monteiro**, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. *Code-Level Model Checking in the Software Development Workflow.* In Proceedings of the 42nd International Conference on Software Engineering (ICSE'20). ACM, New York, NY, USA, 2020.

Media

viii **Felipe R. Monteiro**. *Verificação Formal e seu Papel no Desenvolvimento de Sistemas Cyber-Físicos Críticos.* Eldorado, 2018. Available at Blog Instituto Eldorado[1].

[1] `https://bit.ly/35k79AT`

Contents

Abstract **iv**

Acknowledgments **v**

Publications **vi**

1 Introduction **1**

1.1 Problem & Motivation 1

1.2 Objectives 2

1.3 Contributions 2

1.4 Outline 3

2 Background Theory **4**

2.1 Satisfiability Modulo Theories 4

2.1.1 Arrays and Tuples 4

2.2 Bounded Model Checking 5

2.3 ESBMC Architecture 6

3 Related Work **8**

4 SMT-based Bounded Model Checking of C++ Programs **12**

4.1 Templates 12

4.2 Standard C++ Libraries 17

4.2.1 Core Language for Standard Containers 18

4.2.2 Sequential Containers 19

4.2.3 Associative Containers 23

4.2.4 Correctness 26

4.3 Inheritance & Polymorphism 26

4.4 Exception Handling 30

5 Experimental Evaluation **36**
5.1 Objectives 36
5.2 Description of the Benchmarks 36
5.3 Experimental Setup 37
5.4 Experimental Results 38
5.5 Discussion 41

6 Conclusions **44**

Bibliography **47**

A Experimental Data **56**

List of Tables

3.1 Related work comparison. 8

4.1 Overview of the C++ Operational Model. 18
4.2 Rules to connect `throw` expressions and `catch` blocks. 35

A.1 ESBMC *v*2.0 experimental data. 57
A.2 DIVINE *v*4.0.22 experimental data. 58
A.3 LLBMC *v*2013.1 experimental data. 59

List of Figures

2.1 ESBMC architectural overview. 6

4.1 Function template example. 15
4.2 Example of IR creation. 16
4.3 The program of Fig. 4.1 in SSA form. 17
4.4 Operational model illustration of the STL sequential containers. 20
4.5 Operational model illustration of the STL associative containers. 23
4.6 `Vehicle` class hierarchy UML diagram. 27
4.7 C++ program using a simplified version of the UML diagram in Fig. 4.6. The program nondeterministically cast derived class to a base class (either `Vehicle` or `Motorcycle` to `Car`). The goal is to check if the correct `number_of_wheels`() is called, from the base class. 28
4.8 GOTO instructions generated for the program in Fig. 4.7. 29
4.9 The program of Fig. 4.7 in SSA form. 30
4.10 Try-catch example of throwing an integer exception. 31
4.11 Try-catch conversion to GOTO instructions. 32
4.12 The program of Fig. 4.10 in SSA form. 32

5.1 Comparison of verification results between ESBMC *v*2.0, DIVINE *v*4.0.22, and LLBMC *v*2013.1. 38
5.2 Comparison of verification results between ESBMC *v*2.0, DIVINE *v*4.0.22, and LLBMC *v*2013.1 regarding general-purpose benchmarks in C++03. 41
5.3 Comparison of accumulative verification time and accumulative memory consumption among ESBMC *v*2.0, DIVINE *v*4.0.22, and LLBMC *v*2013.1 throughout the verification process of all benchmarks. 42

Chapter 1

Introduction

In this work, we describe and evaluate a novel SMT-based BMC approach to verify C++ programs using an operational model, an abstract representation of the Standard C++ Libraries (SCL) that reflects their semantics. We integrate this approach into the Efficient SMT-Based Context-Bounded Model Checker (ESBMC) [29, 43–45] and formalize how ESBMC handles templates, sequential and associative containers, inheritance, polymorphism, and exception handling using a decidable fragment of the first-order logic [16].

1.1 Problem & Motivation

The main problem this work aims to solve is how to verify the memory safety properties of C++ programs automatically. For more than 30 years now, memory safety issues in low-level programming languages such as C or C++ have been one of the major sources of security vulnerabilities [90]. For instance, the Microsoft Security Response Center reported that approximately 70% of their security vulnerabilities each year are due to memory safety issues in their C and C++ code [66]. Beyond memory safety, undefined behavior (e.g., signed integer overflow) also represents another crucial source of errors in C and C++ programs that could potentially lead to security issues [53].

Software verification plays a vital role in ensuring the overall product reliability. Over the last 15 years, formal techniques have been dramatically evolved [26], its adoption in industry has been growing [22, 27, 36, 83], and several tools to formally verify C programs have been proposed [12]; however, there exist only a few attempts with limited success to cope with the complexity of C++ program verification [4, 15, 40, 50, 69–71, 78, 87, 91–93, 99]. The main challenge here is to support sophisticated features that the C++ programming language offers, such as templates, sequential and associative template-based containers, inheritance, polymorphism, and exception handling. At the same time, to be attractive for mainstream software development, C++ verifiers must handle large programs, maintain high speed and soundness, in addition to supporting legacy designs.

1.2 Objectives

This work aims to demonstrate the efficiency and effectiveness of formal verification techniques to prove the absence of memory safety and undefined behavior issues in C++ programs. This general goal is correlated with the following specific ones:

i. Provide a logical formalization of essential features that the C++ programming language offers, such as templates, sequential and associative containers, inheritance, polymorphism, and exception handling.

ii. Provide a set of abstractions to the Standard C++ Libraries (SCL) that reflects their semantics, in order to enable the verification of functional properties related to the use of these libraries.

iii. Extend an existing verifier to handle the verification of C++ programs based on *(i)* and *(ii)* and evaluate its efficiency and effectiveness in comparison to similar state-of-the-art approaches.

1.3 Contributions

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 [14]. Formal verification of source code can have a significant positive impact on the quality of code. In particular, formally verified specifications of code provide precise, machine-checked documentation for developers and consumers of a codebase. They improve code quality by ensuring that the program's *implementation* reflects the developer's *intent*. Unlike testing/fuzzing, which can only validate code against a set of concrete inputs, formal proof can assure that the code is both secure and correct for all possible inputs [13, 97]; however, it is important to highlight that one does not invalidate the other and both approaches are fundamental to achieve higher levels of software quality.

In this context, R. Gadelha et al. [42, 80] initiated the support of the formal verification of C++ programs in ESBMC. The previous work focused on the support for templates (partially), sequential containers, and the implementation of exception handling in ESBMC. We describe and evaluate novel approaches to handle sequential and associative containers, in addition to the formalization of ESBMC's mechanisms to handle inheritance, polymorphism, and all its throw & catch exception rules. ESBMC can check for undefined behaviors and memory safety issues such as under- and overflow arithmetic, division-by-zero, pointer safety, array out-of-bounds violations, and user-defined assertions. In addition to these properties, the combination of ESBMC and the C++ operational models enables us to verify specific properties related to C++ structures (e.g., functional properties of standard containers), via pre- and postconditions. ESBMC also provides specific strategies to handle exceptions in C++ programs (e.g., exception specification for functions and methods), which previous approaches could not handle [15, 40, 78].

Precisely, the major contributions of this work are:

1. the formal description of how ESBMC handles primary template, explicit-template specialization, and partial-template specialization;
2. the operational model structure to handle new features from the SCL (e.g., sequential and associative template-based containers);
3. the formalization of the ESBMC's engine to handle inheritance & polymorphism;
4. the formalization of all throw & catch exception rules supported by ESBMC;
5. the expressive set of publicly available benchmarks designed specifically to evaluate software verifiers that target the C++ programming language;
6. the comparative evaluation of state-of-the-art software model checkers on the verification of C++ programs. We compare the proposed approach against the LLBMC [40], a bounded model checker based on SMT solvers, and DIVINE [4], an explicit-state model checker, both for ANSI-C and C++ programs. Our experimental evaluation contains a broad set of over $1,500$ benchmarks, where ESBMC reaches a success rate of 84.27%, which significantly outperforms both LLBMC and DIVINE.

1.4 Outline

The remainder of this document is organized as follows. Section 2 gives a brief introduction to BMC and the ESBMC architecture and describes the essential background theories of the SMT solvers. Section 3 discusses the related work. In Section 4, we describe the main contributions: Section 4.1 presents the proposed approach to support templates; Section 4.2 presents the operational model to replace the SCL in the verification process; Section 4.3 presents the formalization of ESBMC's mechanism to support inheritance & polymorphism features, respectively; and Section 4.4 formally describes the exception handling approach in ESBMC. Furthermore, in Section 5, we present the results of the experimental evaluation using over $1,500$ C++ benchmarks and extra data is presented in Appendix A. The evaluation also compares the experimental results to other state-of-the-art C++ model checkers. Finally, Section 6 contains the conclusion and future directions.

Chapter 2

Background Theory

ESBMC is a bounded model checker built on an improved version of the front-end of CBMC to generate the VCs for a given ANSI-C/C++ program, and encode them using different background theories (i.e., linear integer and real arithmetic, and bitvectors) and SMT solvers (i.e., Boolector [75], Z3 [33], Yices [38], MathSAT [23], and CVC4 [5]). 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) [6–12, 86]; in particular, it was recently ranked at the top three verifiers in the overall ranking of SV-COMP 2018.[1]

2.1 Satisfiability Modulo Theories

SMT decides the satisfiability of a fragment of quantifier-free first-order formulae using a combination of different background theories and, thus, generalizes propositional satisfiability by supporting uninterpreted functions, linear and non-linear arithmetic, bit-vectors, tuples, arrays, and other decidable first-order theories. Given a theory 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 [17]. Given a set $\Gamma \cup \{\psi\}$ of formulae over τ, we say that ψ is a τ-consequence of Γ, and write $\Gamma \models_\tau \psi$, if and only if every model of $\tau \cup \Gamma$ is also a model of ψ. Checking $\Gamma \models_\tau \psi$ can be reduced in the usual way to checking the τ-satisfiability of $\Gamma \cup \{\neg\psi\}$.

2.1.1 Arrays and Tuples

The most important theories for ESBMC are the array and tuple theories, which are used to model the sequential container data structures and objects, respectively. The array theories of SMT solvers are typically based on the McCarthy axioms [62]. The function *select(a, i)* denotes the value of an

[1] `https://sv-comp.sosy-lab.org/2018/results/results-verified/`

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 [5, 18, 33]:

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

Array bounds checks need to be encoded separately, as the array theories assume arrays of unbounded size. However, arrays in software are of bounded size.

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

2.2 Bounded Model Checking

In BMC, the program to be analyzed is modeled as a state transition system, which is extracted from the control-flow graph (CFG) [72]. 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 [14]. The associated model checking problem is formulated by constructing the following logical formula:

$$\psi_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg\phi(s_i), \tag{2.1}$$

given that ϕ is a safety property, I is the set of initial states of M and $T(s_j, s_{j+1})$ is the transition relation of M between steps j and $j+1$. Hence, $I(s_0) \wedge \bigwedge_{j=0}^{i-1} T(s_j, s_{j+1})$ represents the executions of M of length i and the formula 2.1 can be satisfied if and only if, for some $i \leq k$, there exists a reachable state at step i in which ϕ is violated. If the formula 2.1 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 $s_0, s_1, \cdots, s_k$ with $s_0 \in S_0$ and $T(s_i, s_{i+1})$ with $0 \leq i < k$.

If the formula 2.1 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, i.e., if we can ensure that we have already explored all the relevant behavior of

the system, and searching any deeper only exhibits states that have already been verified [58].

2.3 ESBMC Architecture

ESBMC is a mature, permissively licensed open-source context-bounded model checker for the verification of C++, single- and multi-threaded C programs [28, 68, 77]. It can verify both predefined safety properties (e.g., bounds check, pointer safety, overflow) and user-defined program assertions automatically. Its development started in 2008 on top of the CProver framework [24], but almost all components have been re-designed and re-implemented in subsequent years, including the basic data structures, front-end, symbolic execution, memory model, and back-end. By default, ESBMC takes a ANSI-C or C++ program and checks for array bounds violations, divisions by zero, pointer safety (incl. alignment), and all user-defined properties. It has options to check for overflows, memory leaks, deadlocks and data-races, and to choose between a fixed- or (IEEE) floating-point arithmetic. ESBMC has also been extended to localize faults in single- and multi-threaded programs [31, 32]. Fig. 2.1 shows the ESBMC architecture [29, 43, 48]. White rectangles represent input and output; gray rectangles represent the steps of the verification.

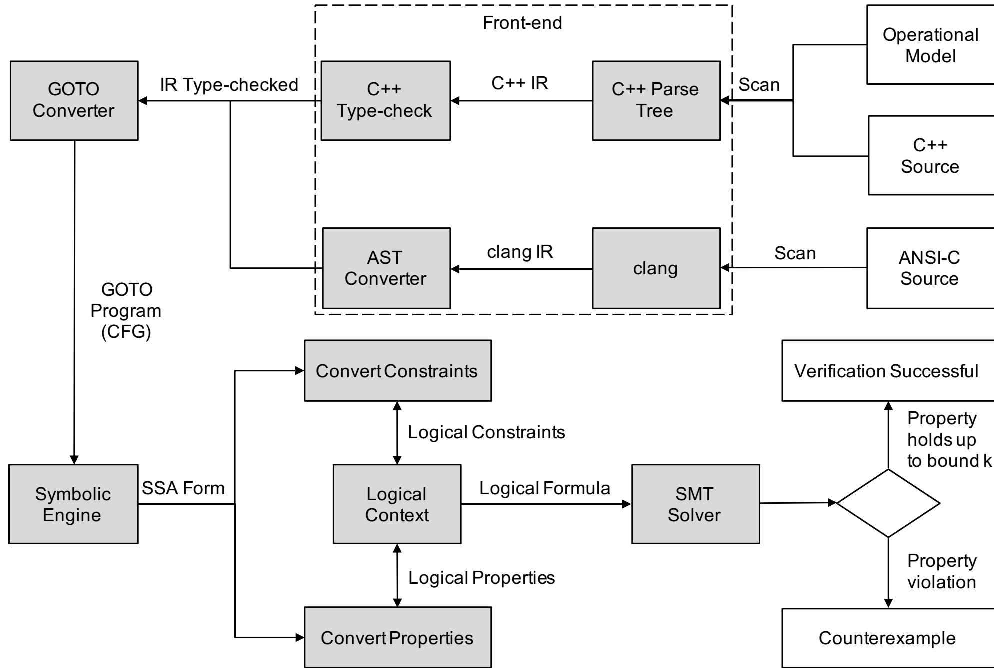


Figure 2.1: ESBMC architectural overview.

Front-end. The first steps are the source code parser and the type-check, which are language-specific. The parser for C++ is heavily based on the GNU C++ Compiler (GCC) [41], which allows

ESBMC to find most of the syntax errors already reported by GCC. For ANSI-C, ESBMC now uses clang [60], a state-of-the-art compiler suite for C/C++/ObjectiveC/ObjectiveC++ widely used in industry [65], as its front-end. As developers, we thus avoid the need to maintain a separate front-end, but this approach also brings many advantages for users: *(i)* ESBMC provides compilation error messages as expected from an industrial-strength tool; *(ii)* ESBMC leverages clang's powerful static analyzer to provide meaningful warnings when parsing the program; *(iii)* clang can simplify the input program (e.g., calculate `sizeof` expressions, evaluate static asserts), which simplifies the analysis of the code. Note that ESBMC uses clang's API to access and traverse the program abstract syntax tree (AST), without having details of the input program compiled away, which differs from other verifiers (e.g., LLBMC [40]) that rely on the LLVM bitcode.

Control-flow Graph (CFG) Generator. On type-checking for C++ programs, the code is statically analyzed, which includes assignment checks, type-cast checks, pointer initialization checks, function call checks and template instantiation. By the end of the type-check, the Intermediate Representation (IR) creation is completed and will be used by the GOTO converter to generate the GOTO program. For ANSI-C, the CFG generator takes the program AST from clang and transforms it into an equivalent GOTO program: a simplified representation that consists only of assignments, conditional and unconditional branches, assumes, and assertions. In particular, this step eliminates all `for`, `while`, `do-while` and `switch` statements. It also adds checks for division by zero and out-of-bounds access (and for integer and floating-point overflow [46], if enabled).

Symbolic Execution Engine. ESBMC then symbolically executes the GOTO program: it unrolls loops k times, generates the static single assignments (SSA) form of the unrolled program, and derives all the safety properties to be checked by the SMT solver. This step also inserts pointer safety checks for dynamically allocated memory, if they are enabled. Note that this can only be done after unrolling because the pointer analysis needs to know the maximum set of dynamically allocated structures. ESBMC aggressively simplifies the program to generate small SSA sets, using constant folding and various arithmetic (including floating-point) simplifications.

SMT back-end. ESBMC's SMT back-end supports five solvers: Boolector (default), Z3, MathSAT, CVC4 and Yices. The back-end is highly configurable and allows the encoding of quantifier-free formulas with support for bitvectors, arrays, tuple, fixed-point and floating-point arithmetic (all solvers), and linear integer and real arithmetic (all solvers but Boolector). We use the back-end to encode the SSA form of the program into a quantifier-free formula and check satisfiability of $C \wedge \neg P$, where C is the set of constraints and P is the set of properties. If the formula is SAT, the program contains a bug: ESBMC will generate a counterexample with the set of assignments that lead to the property violation.

Chapter 3

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 and there exist more stable verification tools written for other programming languages, such as C [12]. This conversion, however, can unintentionally introduce or hide errors in the original program; in particular, the verification of the converted program may present different results if compared to the verification of the original C++ program, unless we check the equivalence of both the original and the modified program [52], which can become undecidable in the presence of unbounded memory usage.

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 templates, and end up neglecting other features of equal importance, such as the verification of the STL [82, 88] or exceptions [64]. Table 3.1 shows a comparison among other studies available in the literature and our proposed approach.

Table 3.1: Related work comparison.

Related work	Conversion to intermediate languages	C++ Programming Language			
		Templates	Standard Template Libraries	Inheritance & Polymorphism	Exception Handling
Merz et al. [40]	LLVM	Yes	Yes	Yes	No
Blanc et al. [15]	No	Yes	Yes	No	No
Prabhu et al. [78]	ANSI-C	Yes	Not mentioned	Yes	Yes
Clarke et al. [24]	No	Yes	No	No	No
Baranová et al. [4]	LLVM	Yes	Yes	Yes	Yes
ESBMC *v*2.0 [80]	**No**	**Yes**	**Yes**	**Yes**	**Yes**

Merz, Falke, and Sinz [40, 64] describe LLBMC, a tool that uses BMC to verify C++ programs.

The tool first converts the program into LLVM intermediate representation, using clang [59] as an off-the-shelf frontend. This conversion removes high-level information about the structure of C++ programs (e.g., the relationship between classes), but 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 before bit-blasting and passed to an SMT solver for verification. The tool does not verify programs with exception handling, which makes it difficult to realistically verify C++ programs, 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 clang generates. For instance, a symbolic virtual machine built on top of the LLVM compiler, named as KLEE [20], still uses an old version of LLVM (v3.4) due to the major effort to update its internal structure.

Blanc, Groce, and Kroening [15] 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 [25]. SATABS is a verification tool for C and C++ programs that supports classes, operator overloading, references, and templates (but without supporting partial specification). The authors show that, in order to verify the correctness of a program, it is sufficient to use an operational model by proving that, if the pre- and postconditions hold, the implementation model also holds. The approach is efficient in finding trivial errors in C++ programs. The preconditions are modeled to verify the library containers using an operational model similar to the model used by the ESBMC tool for the same purpose. Regarding the operational model, the authors present only the preconditions, while our operational model verifies preconditions and replicates the behavior of the STL, which increases the range of applications that can be properly verified by the tool (i.e., postconditions).

Prabhu, Maeda, and Balakrishnan [78] present an inter-procedural exception analysis and transformation framework for C++ programs, which records the flow of the program created by exceptions and creates an exception free C program. The creation of exception-free programs starts with the generation of a control-flow graph, called Inter-procedural Exception Control-Flow Graph (IECFG). The IECFG is then analyzed by an algorithm developed by the authors that models all possible exceptions, which can connect the *catch* blocks using a compressed representation, called *signed-typeset*. The result of the modeling is then used to generate the exception-free C program, which simulates the behavior of throwing and catching exceptions by assigning the thrown object to a local object (i.e., assigned to the object of the declaration of the *catch* block), and by the use of

GOTO instructions. At the end of the conversion process, the C program is checked using the F-SOFT [56] tool. For the experimental evaluation, 18 C++ programs and 4 commercial applications were used. The verification of these benchmarks focuses on only two properties: "no throw" (the percentage of code that does not generate exceptions) and "no leak" (the number of memory leaks in *try* blocks) [78]. The authors do not indicate whether this approach is able to verify other types of errors in addition to the two described properties. Connection rules and formal verification using exception handling are also presented. The technique described by the authors is similar to ours, including the use of GOTO instructions to model jumps to catch blocks. The difference is the number of properties our tool can verify: we can not only check the paths that throw exceptions and memory leaks in try blocks, but we also check for property violations in the paths where exceptions were thrown. Futhermore, the authors make no mention of exception specialization or terminate/unexpected handling.

Clarke, Kroening, and Lerda [24] present CBMC, which implements BMC for C/C++ programs using SAT/SMT solvers. CBMC uses its own parser, based on Flex/Bison [29], 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 will be processed by the symbolic execution engine. 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 [85]. 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. Chapter 5). In particular, CBMC has problems instantiating template correctly, and lacks support for exception specialization and terminate/unexpected functions.

Baranová et al. [4] 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[1] and DVE[2]). 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 [59] as front-end, DIVINE translates C++ programs into the LLVM intermediate representation, thereby applying its own implementation of the C and C++ standard libraries, in order

[1] http://www.uppaal.org
[2] http://divine.fi.muni.cz/index.html

to ensure a consistent translation. Nonetheless this translation process might cause some irregularities to the verification process, once it looses 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 [88] 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 [82]. The authors also claim DIVINE as the first model checker that is able to verify exception handling in C++ programs, as opposed to what has been stated by Ramalho et al. [80]. However, ESBMC $v1.23$ (i.e., the version used by Ramalho et al. [80]) is able to correctly verify the example presented by Ročkai, Barnat and Brim [88], generating and verifying 10 VCs in less than one second. In fact, our experimental evaluation shows that ESBMC outperforms DIVINE in handling exception as well as for the support of standard containers, inheritance, and polymorphism (cf. Chapter 5).

Marjamäki [61] developed Cppcheck, a static analyzer for C/C++ primarily focuses on detecting undefined behaviors using multiple checkers for specific violations, e.g., dead pointers, division by zero, invalid bit shift operands, invalid conversions, null pointer dereferences, out of bounds violations, uninitialized variables, among others. Firstly, it preprocesses the source files and produces a token list, which is then converted into a syntax tree. The analyzer also keeps a symbol database with additional information about the source code. Cppcheck performs a data-flow analysis before running specific checkers, in order to assign each token a list of possible values for all variables. Finally, each checker uses the information from previous steps to look for a particular vulnerability (e.g., integer overflows). Even though the major goal of the tool is to report as few false positive as possible, Ågren [79] points that the tool could still lead to false positives due to lack of support for function pointers, macro expansion, or inheritance. The prime difference between ESBMC and Cppcheck is that Cppcheck is a bug-finding tool, and ESBMC can not only find bugs but can also be used to prove correctness, either by fully unrolling all loops in the program or by using k-induction [47,48]. Additionally, ESBMC uses a bit-accurate verification engine to check for arithmetic under- and overflow while Cppcheck is unable to precisely check for such properties [3,67,73]. Nevertheless, Cppcheck was applied to major projects including Debian Automated Code Analysis (DACA) [51] and OpenOffice.org [57].

Chapter 4

SMT-based Bounded Model Checking of C++ Programs

In this chapter, we start by describing the formalization behind the support for C++ templates, its application, and how ESBMC handles them. Importantly, the current support of templates is the base for the STL. We then describe our operational model, a simplified version of the SCL, specially developed for verification purposes. We focus on the description of the containers provided by STL, presenting the basic operations that reproduce the behavior of the standard. We also describe how ESBMC deals with inheritance and polymorphism in C++ programs. Lastly, we formally define the exception handling in ESBMC, including the description of the connection between `throw` and `catch` statements, exception specification and, `terminate` and `unexpected` handlers.

4.1 Templates

The concept of templates in the C++ programming language is more than twenty years old [96], and represents one of its most essential features; in the early 1990s, templates were already documented in the literature [39], and their usage has grown ever since. Templates are generally used to define functions or classes of generic types, which can be later instantiated with a specific data type. Reusability is the main advantage of the usage of templates since it is not necessary to write a different version of classes or functions for each type used in the C++ program. In addition to the primary templates, C++ allows explicit template specialization as an alternative implementation to specialize a (class or function) template to a specific data type [55]. Similarly, partial template specialization is also allowed, i.e., where only some of its template arguments are specialized while others remain generic [55]. 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 [55]. Indeed, the current support for C++ templates in ESBMC is one of the key features that enable it to properly verify STL, SCL, and real-world C++ programs (*cf.*, Chapter 5), even though it is still a work-in-progress.

Templates are not run-time objects [89] when a C++ program is compiled, classes and functions are generated from templates, and 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 function or class is no longer a template. Hence, at the end of the type-checking phase, all templates are entirely 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 more straightforward and 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.

Regarding template instantiation, the implementation in ESBMC is based on the formalization previously presented by Siek and Taha [84] who introduced the first proof of type safety of the template instantiation process for C++03 programs. To properly describe the template instantiation process in ESBMC, we formally define the syntactic domains Π, T, S, $\mathcal{A}$, $\mathcal{N}$, and $\mathcal{K}$ as follows:

$$
\begin{aligned}
\Pi &:= \pi \\
T &:= \tau \\
S &:= s \mid s_e \mid s_p \\
\mathcal{A} &:= a \mid \mathbb{A} \\
\mathcal{N} &:= \texttt{name} \mid \mathcal{I}.\texttt{name} \mid G.\texttt{name} \\
\mathcal{K} &:= k \mid \mathcal{I}.k \mid G.k \mid \texttt{class} \mid \texttt{func}
\end{aligned}
$$

In this context, π, τ, and s are classes of variables of type template instantiations Π, templates T, and template specializations S, respectively. We abuse the notation s_e to denote an explicit template specialization and s_p to indicate a partial template specialization. Here, τ is also referred to as the primary template. The static domain $\mathcal{A}$ represents the set of all template arguments, which consists of two specific subsets $G \subseteq \mathcal{A}$ for generic tokens and $D \subseteq \mathcal{A}$ for data types. Each template instantiation in the program is represented by a tuple $\pi = \langle \mathcal{N}, \mathcal{K}, \mathbb{A}_\pi \rangle$, where $\mathcal{N}$ is the template name, $\mathcal{K}$ is the kind of template (i.e., either `class` for class templates or `func` for function templates), and $\mathbb{A}_\pi = \{a_1, ..., a_m | a \in D\}$ is the set of m template arguments used on a certain instantiation. Similarly, each template definition in the program is represented by a tuple $\tau = \langle \mathcal{N}, \mathcal{K}, \mathbb{A}_\tau, \mathbb{S}_\tau \rangle$, where $\mathcal{N}$ is the template name, $\mathcal{K}$ is the kind of template (i.e., either `class` or `func`), $\mathbb{A}_\tau = \{a_1, ..., a_n \mid a \in G\}$ is the set of n template arguments, and $\mathbb{S}_\tau = \{s_1, ..., s_k \mid s \in S\}$ is the set of k template specializations for the primary template τ. Each $s_1, ..., s_k$ represents a template specialization, which is defined by the tuple $s = \langle \mathcal{K}, \mathbb{A}_s \rangle$, where $\mathcal{K}$ is the kind of template (i.e., either `class` or `func`) and $\mathbb{A}_s$ is

defined differently for each specialization. For explicit function specialization $s_e = \langle \texttt{func}, \mathbb{A}^{\texttt{func}}_{s_e} \rangle$, the set of template arguments is defined as $\mathbb{A}^{\texttt{func}}_{s_e} = \{a_1, ..., a_n \mid a \in D\}$; since C++ does not allow partial template specialization of function templates, all arguments must be data types. Similarly, for explicit class specialization $s_e = \langle \texttt{class}, \mathbb{A}^{\texttt{class}}_{s_e} \rangle$, the set of template arguments is defined as $\mathbb{A}^{\texttt{class}}_{s_e} = \{a_1, ..., a_n \mid a \in D\}$. For partial template specialization $s_p = \langle \texttt{class}, \mathbb{A}^{\texttt{class}}_{s_p} \rangle$, the set of template arguments is defined as $\mathbb{A}^{\texttt{class}}_{s_p} = \{a_1, ..., a_n \,|\, \exists! a \in G \wedge \exists! a \in D\}$, since there must be at least one remaining generic token and at least one specialized data type. Template specializations do not carry on extra information (e.g., `name`) because they are already linked to their primary template definitions during the template creation process. Furthermore, the implementation of classes and functions are omitted from this formalization, because they are not relevant for the instantiation process.

Based on such domains, we must define a 2-arity predicate $\mathcal{M}(\pi, \tau)$, which evaluates whether a given template instantiation matches a template definition based on its `name` and type k, as described by Eq. (4.1). Furthermore, we declare the 2-arity function $\lambda : T^* \times \Pi \longmapsto S$, which selects the most specialized template in τ given a template instantiation π as described by Eq. (4.2). The case where there is no template specialization suitable for π (i.e., $\varnothing$) is an indication to select the primary template definition. To read function λ, we must introduce the notion of "*most specialized*", which is represented by the operator $\succeq$. In this case, given a template instantiation π and two template specializations $s_{\texttt{M}}$ and s, the expression $(s_{\texttt{M}}, \mathbb{A}_\pi) \succeq (s, \mathbb{A}_\pi)$ indicates that the template specialization $s_{\texttt{M}}$ is more specialized for π than s, based on the set of arguments $\mathbb{A}_\pi$, i.e., $\mathbb{A}_{s_{\texttt{M}}}$ matches more template arguments in $\mathbb{A}_\pi$ than the set of template arguments $\mathbb{A}_s$. Function λ is crucial to ESBMC, since it must select the most specialized template, which matches the given template arguments $\mathbb{A}_\pi$ [55, 84].

$$\mathcal{M}(\pi, \tau) \stackrel{\text{def}}{=} \begin{cases} \top, & \pi.\texttt{name} = \tau.\texttt{name} \wedge \pi.k = \tau.k \\ \bot, & \text{otherwise} \end{cases} \tag{4.1}$$

$$\lambda(\pi, \tau) \stackrel{\text{def}}{=} \begin{cases} s_{\texttt{M}}, & \forall s \in \mathbb{S}_\tau \cdot (s_{\texttt{M}}, \mathbb{A}_\pi) \succeq (s, \mathbb{A}_\pi) \\ \varnothing, & \text{otherwise} \end{cases} \tag{4.2}$$

$$\begin{aligned} \mathcal{L}(\pi, \tau_1, ..., \tau_q) := ite(&\mathcal{M}(\pi, \tau_1), \tau_\pi = \tau_1, \\ & ite(\mathcal{M}(\pi, \tau_2), \tau_\pi = \tau_2, \\ & \ldots \\ & ite(\mathcal{M}(\pi, \tau_q), \tau_\pi = \tau_q, \tau_\pi = \varnothing) \ldots) \\ & \wedge \tau_\pi \neq \varnothing \\ & \wedge s = \lambda(\pi, \mathbb{S}_{\tau_\pi}) \\ & \wedge ite(s = \varnothing, \tau_\pi, s) \end{aligned} \tag{4.3}$$

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 π is found, ESBMC firstly identifies which type of template k it is dealing with (i.e., either `class` or `func`) and which

template arguments $\mathbb{A}_\pi$ is used, setting the tuple $\pi = \langle \texttt{name}, k, \mathbb{A}_\pi \rangle$. It then searches whether an IR of that type was already created, i.e., whether π has been previously instantiated. If so, no new IR is created; this avoids duplicating the IR, thus reducing the memory requirements of ESBMC. If there exists no IR of that type, a new IR is created, used in the instantiation process, and saved for possible future searches. In order to create a new IR, ESBMC must select the most specialized template for π; therefore, ESBMC performs another search in the IR to select the proper template definition $\tau_\pi = \langle \texttt{name}, k, \mathbb{A}_{\tau_\pi}, \mathbb{S}_{\tau_\pi} \rangle$ based on the predicate $\mathcal{M}(\pi, \tau)$. ESBMC then checks whether there exists a (partial or explicit) template specialization in $\mathbb{S}_{\tau_\pi}$, which matches the set of data types in the instantiation. If ESBMC does not find any template specialization s, which matches the template arguments, it will select the primary template definition τ, as described by Eq. (4.3). Once the most specialized template is selected from $\mathcal{L}(\pi, g_1, ..., g_q)$, ESBMC performs a transformation to replace all generic types for the data types specified in the instantiation $\mathbb{A}_\pi$; this transformation is necessary because, as stated previously, at the end of the C++ type-checking phase all templates are removed.

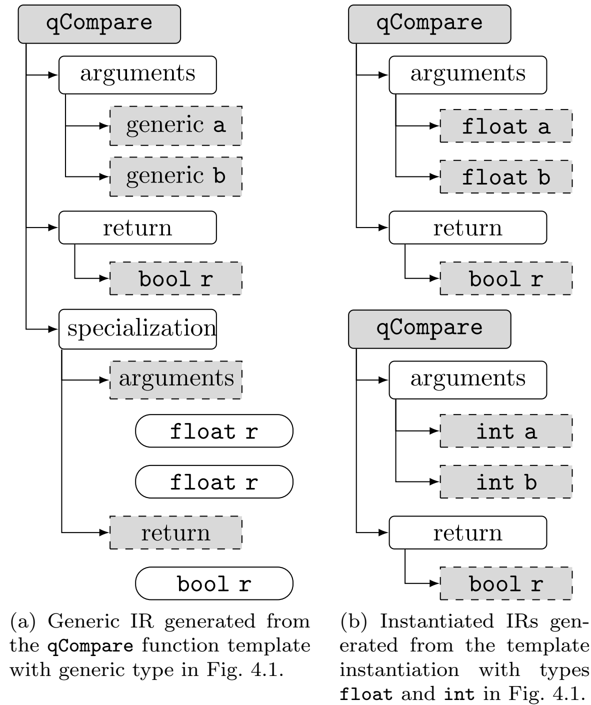
```
1  #include<cassert>
2  using namespace std;
3
4  // template creation
5  template <typename T>
6  bool qCompare(const T a, const T b) {
7    return (a > b) ? true : false;
8  }
9
10 template <typename T>
11 bool qCompare(T a, T b) {
12   return (a > b) ? true : false;
13 }
14
15 // template specialization
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 4.1: Function template example.

In order to concretely demonstrate the instantiation process in ESBMC, Fig. 4.1 shows an example of function templates usage, which is based on the example `spec29` extracted from the GCC test suite[1]. First, 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. Template instantiation happens when the template is used. In Fig. 4.1, the template is instantiated twice (lines 23 and 24). In fact, it is also possible to determine the type implicitly (line 23) or explicitly

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

(line 24). In implicit instantiation, the data type is determined by the types of the used parameters, while in the explicit instantiation, the data type is determined by the value passed between the $<$ and $>$ symbols.



(a) Generic IR generated from the `qCompare` function template with generic type in Fig. 4.1.

(b) Instantiated IRs generated from the template instantiation with types `float` and `int` in Fig. 4.1.

Figure 4.2: Example of IR creation.

Fig. 4.2 shows the generic IR and the instantiated IRs generated from the code in Fig. 4.1. Fig. 4.2a shows the generic IR generated from the `qCompare` function template and its specialization, while Fig. 4.2b 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 Fig. 4.2a. Note that the data type is labeled as `generic`, which means that the type is generic. In Fig. 4.2b, the data types that were previously labeled as `generic` are now labeled 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 (see Fig. 4.2a) is discarded.

After the template instantiation, the verification process resumes, as described by Cordeiro et al. [30]. As an illustrative example, for the program in Fig. 4.1, ESBMC generates the SSAs shown

```
1 a1 = 1.5f
2 b1 = 2.5f
3 return_qcompare1 = (b1 > a1)? TRUE : FALSE
4 a2 = 1
5 b2 = 2
6 return_qcompare2 = (a2 > b2)? TRUE : FALSE
```

Figure 4.3: The program of Fig. 4.1 in SSA form.

in Fig. 4.3. Note that variable declarations and return statements are removed. The SSA form only consists of conditional and unconditional assignments; in addition, assertions are removed. After this transformation, ESBMC builds the constraints and properties, as shown in Eqs. (4.4) and (4.5) in terms of background theories of the SMT solvers. Then, ESBMC performs simplifications on $\mathcal{C}$ and $\mathcal{P}$ formulae, in order to remove functionally redundant expressions and redundant literals. Finally, the formula $\mathcal{C} \land \neg\mathcal{P}$ is given to an SMT solver to verify satisfiability.

$$\mathcal{C} := \left[\begin{array}{l} a_1 = 1.5f \land b_1 = 2.5f \\ \land\ return_qcompare_1 = ite\,(b_1 > a_1, 1, 0) \\ \land\ a_2 = 1 \land b_2 = 2 \\ \land\ return_qcompare_2 = ite\,(a_2 > b_2, 1, 0) \end{array}\right] \tag{4.4}$$

$$\mathcal{P} := \left[\begin{array}{l} return_qcompare_1 = \top \\ \land\ return_qcompare_2 = \bot \end{array}\right] \tag{4.5}$$

4.2 Standard C++ Libraries

The C++ programming language offers a collection of powerful libraries, called SCL, to provide most of the functionalities required by the programmer [89]. In particular, a significant set of this collection of libraries, called STL, relies on templates to provide flexibility for the development of a C++ program. However, the direct verification of the SCL unnecessarily complicates the verification of a C++ program, as it contains code fragments irrelevant for verification (e.g., code fragments to write messages on the screen and optimized assembly code) [70, 80].

To reduce verification complexity, ESBMC uses a simplified Standard C++ Library called the C++ Operational Model (COM), which covers the essential behaviors of the SCL [55]. A similar technique, proposed by Blanc, Groce, and Kroening [15], has been used to verify preconditions on programs. However, ESBMC extends that approach by checking the preconditions and simulating the behavior of the SCL (i.e., checking postconditions), which improves the effectiveness of the model checker, as shown in our experimental evaluation (cf., Chapter 5). Our COM removes all the irrelevant code for verification, while adding checks, e.g., if the iterators are valid before performing the operation.

In order to be a genuine representation of the SCL, the COM presented shares the same structure of the standard, as can be seen in Table 4.1. The COM consists of eight groups of libraries: (i) *C*

Standard Library, which consists of C libraries with the following differences: the headers have no extension and every element of the library is defined within the *std* namespace; importantly, since ESBMC uses a different front-end (as shown in Fig. 2.1), we have to build a representation of the C libraries into the COM; otherwise, ESBMC would not recognize the library methods and fail to parse the C++ programs; (*ii*) *Streams and Input/Output*, which consists of libraries that provide input and output functionality using streams; (*iii*) *Numeric*, which offers mechanisms to perform numerical operations; (*iv*) *Language support*, which consists of types and functions for exception handling (e.g., `exception`), functions to manage dynamic allocation (e.g., `new`) among other base functionalities; (*v*) *Strings*, which provide string types and character traits (e.g., `string`); (*vi*) *Localization*, which offers functions related to basic localization routines; (*vii*) *General*, which consists of general purpose libraries that provide, for instance, functions designed to be used on ranges of elements (e.g., `algorithm`); and (*viii*) *Containers*, which provide types used to store objects that can be accessed sequentially (e.g., `vector`) or are sorted in order to be quickly searched (e.g., `map`), among other types known as associative containers. Note that most of the aforementioned groups rely on templates to provide their functionalities.

Table 4.1: Overview of the C++ Operational Model.

Standard C++03 Libraries – Operational Model							
C Standard Library	General	Streams & Input/Output	Containers	Language Support	Numeric	Strings	Localization
<cassert>	<algorithm>	<ios>	<bitset>	<exception>	<complex>	<string>	<locale>
<cctype>	<functional>	<iomanip>	<deque>	<limits>	<random>		
<cerrno>	<iterator>	<iosfwd>	<list>	<new>	<valarray>		
<cfloat>	<memory>	<iostream>	<map>	<typeinfo>	<numeric>		
<ciso646>	<stdexcept>	<istream>	<multimap>				
<climits>	<utility>	<ostream>	<set>				
<clocale>		<streambuf>	<multiset>				
<cmath>		<sstream>	<vector>				
<complex>		<fstream>	<stack>				
<csetjmp>			<queue>				
<csignal>							
<cstdarg>							
<cstddef>							
<cstdio>							
<cstdlib>							
<cstring>							
<ctime>							

4.2.1 Core Language for Standard Containers

One of the most challenging part of the COM is the support for the STL. This part is split into four categories: *algorithms*, *containers*, *functors*, and *iterators* [54]. In this work, we focus on the operational model of the sequential and associative containers along with their iterators, and how they are used to verify real-world C++ programs. Particularly, 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. Containers map naturally into SMT array theory; however, the C++03 container implementation is based on a pointer structure that degrades the verification performance [15]. To tackle such a problem, ESBMC applies COM to reduce verification complexity and insert assertions that check for pre- and postconditions related to containers usage.

In order to properly formalize the verification of both sequential and associative containers, we extend the previous core container language presented by Ramalho et al. [80]. The core language defines six syntactic domains, including values V, keys K, iterators I, pointers P, container C and integers $\mathbb{N}$. Here v, k, p, i, c and n are classes of variables of type V, K, P, I, C and $\mathbb{N}$, respectively. We abuse the notation $*i$ to denote the value stored in the underlying container at the position pointed to by the iterator i, $*p$ is the value stored in the p position of the memory. $C.begin$ and $C.end$ are methods that return iterators, which point to the beginning and the ending of a container, respectively. Based on such domains, we also define $P(+|-)P$ as valid pointer operations and $\mathbb{N}(+\mid *\mid\ldots)\mathbb{N}$ as valid integer operations. Thus, the complete container language is defined as follow:

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

Each operation shown in the core container syntax (e.g., $C.insert(I, V)$) is explained in Sections 4.2.2 and 4.2.3.

4.2.2 Sequential Containers

Sequential containers are built into a structure to store elements with a sequential order [35]. In our model, a container c consists of a pointer c_v that points to the underlying container and an integer $size$ that stores the quantity of elements in the container. Similarly, iterators are modeled using two variables: an integer pos, which contains the index value pointed by the iterator in the container and a pointer i_v, which points to the underlying container. In our model, the defined notation $*i$ is equivalent to $(i.i_v)[i.pos]$. Fig. 4.4 gives an overview of our operational model for the STL sequential containers.

All methods from the STL can be expressed as simplified variations of three main operations: insertion ($C.insert(I, V)$), deletion ($C.erase(I)$), and search ($C.search(V)$). As part of the SSA

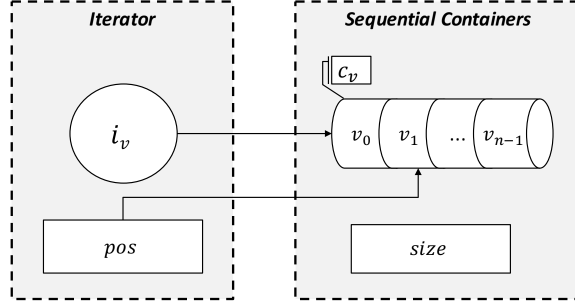


Figure 4.4: Operational model illustration of the STL sequential containers.

transformation, side effects on iterators and containers are made explicit, so that operations return new iterators and containers with the same contents, except for the field that has just been updated. Thus, the translation function $\mathcal{C}$ describes constraints relating the "before" and "after" versions of the respective model variables. Indeed, notations with apostrophe (*e.g.*, c' and i') represent the state of model variables after the respective operation, and simplified notations (*e.g.*, c and i) represent their previous states. Since ESBMC generates quantifier-free formulas, all loops (such as `for` and `while`) are always unrolled and the respective expressions (i.e., $\mathcal{C} \wedge \neg\mathcal{P}$) are thus encoded into SMT. Therefore,

$$\begin{aligned} \forall\, n \cdot (lower_{bound}\ \leq\ n\ \leq\ upper_{bound})\ \Rightarrow \\ select(store(p,\ n,\ v),\ n) = v \end{aligned} \tag{4.6}$$

represents a loop expression, where each value of an array p (i.e., v), from $lower_{bound}$ to $upper_{bound}$ positions, will be selected. As we deal with quantifier-free formulae, Eq. 4.6 actually is a short representation of

$$\begin{aligned} select(store(p,\ lower_{bound},\ v),\ lower_{bound}) = v \\ \dots \\ select(store(p,\ upper_{bound},\ v),\ upper_{bound}) = v \end{aligned} \tag{4.7}$$

Similarly,

$$\begin{aligned} \forall\, n_1 \cdot (lower_{bound}\ \leq\ n_1\ \leq\ upper_{bound})\ \Rightarrow \\ p_2' = store(p_2,\ n_1,\ select(store(p_1,\ n_1,\ v),\ n_1)) \end{aligned} \tag{4.8}$$

also represents a loop expression, where each value of p_1 (i.e., v), from $lower_{bound}$ to $upper_{bound}$ positions, will be stored into p_2'. Eq. 4.8 is a short representation of

$$\begin{aligned} p_2' = store(p_2,\ &lower_{bound}, \\ &select(store(p_1,\ lower_{bound},\ v),\ lower_{bound})) \\ &\dots \\ p_2' = store(p_2,\ &upper_{bound}, \\ &select(store(p_1,\ upper_{bound},\ v),\ upper_{bound})) \end{aligned} \tag{4.9}$$

The statement $c.insert(i, v)$ becomes $(c', i') = c.insert(i, v)$, which has explicit side effects. Here, we increase the container size, move all elements from position $i.pos$ one memory position forward,

and then insert v into the specified position. Therefore[2],

$$\begin{aligned}
&\mathcal{C}((c', i') = c.insert(i, v)) := \\
&\quad c'.size = c.size + 1 \\
&\quad \wedge \forall\, n_1 \cdot (i.pos \;\le\; n_1 \;\le\; c.size) \;\Rightarrow \\
&\qquad\quad c'.c_v = store(c.c_v,\; n_1 + 1, \\
&\qquad\qquad\qquad\qquad select(store(c.c_v,\; n_1,\; v_1),\; n_1)) \\
&\quad \wedge c'.c_v = store(c.c_v,\; i.pos,\; v) \\
&\quad \wedge i'.i_v = c'.c_v \\
&\quad \wedge i'.pos = i.pos
\end{aligned} \tag{4.10}$$

that induces the following properties,

$$\begin{aligned}
&\mathcal{P}((c', i') = c.insert(i, v)) := \\
&\quad v \;\neq\; null \\
&\quad \wedge c.c_v \;\neq\; null \\
&\quad \wedge i.i_v \;\neq\; null \\
&\quad \wedge 0 \;\le\; i.pos \;<\; c'.size \\
&\quad \wedge select(i'.i_v,\; i'.pos) \;=\; v
\end{aligned} \tag{4.11}$$

where *null* represents an uninitialized pointer/object. Thus, we define as preconditions that v and i can not be uninitialized objects as well as $i.pos$ must be within $c'.c_v$ bounds; similarly, we define as postconditions that v was correctly inserted in the position specified by i as well as $c'.c_v$ and $i'.i_v$ are equivalent, i.e., 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 $c.size - 1$. In addition, the main effect of the *insert* method is thus captured by the second equality 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 also 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 [55]. Therefore,

$$\begin{aligned}
&\mathcal{C}((c', i') = c.erase(i)) := \\
&\quad \forall\, n_1 \cdot (i.pos + 1 \;\le\; n_1 \;\le\; c.size - 1) \;\Rightarrow \\
&\qquad\quad c'.c_v = store(c.c_v,\; n_1 + 1, \\
&\qquad\qquad\qquad\qquad select(store(c.c_v,\; n_1,\; v_1),\; n_1)) \\
&\quad \wedge c'.size = c.size - 1 \\
&\quad \wedge i'.i_v = c'.c_v \\
&\quad \wedge i'.pos = i.pos + 1
\end{aligned} \tag{4.12}$$

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

that implicitly induces the following properties,

$$
\begin{aligned}
&\mathcal{P}((c', i') = c.erase(i)) := \\
&\quad i.i_v \neq null \\
&\quad \wedge\, c.c_v \neq null \\
&\quad \wedge\, c.size \neq 0 \\
&\quad \wedge\, 0 \leq i.pos < c.size \\
&\quad \wedge\, select(c'.c_v,\ i'.pos) = select(c.c_v,\ i.pos + 1)
\end{aligned} \tag{4.13}
$$

where we assume as preconditions that i must be a valid iterator that points to a position within the array $c.c_v$ bounds and c must be non-empty; similarly, we assume as postconditions 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,

$$
\begin{aligned}
&\mathcal{C}((c', i') = c.search(v)) := \\
&\quad c'.c_v = c.c_v \\
&\quad \wedge\, c'.size = c.size \\
&\quad \wedge\, i'.pos = 0 \\
&\quad \wedge\, i'.i_v = c'.c_v \\
&\quad \wedge\, \forall\, n \cdot (0 \leq n \leq c'.size - 1) \Rightarrow \\
&\qquad\quad \big(i'.pos = ite(select(c'.c_v, n) = v, n, c'.size)\big)
\end{aligned} \tag{4.14}
$$

that implicitly induces the following properties,

$$
\begin{aligned}
&\mathcal{P}((c', i') = c.search(v)) := \\
&\quad v \neq null \\
&\quad \wedge\, c.c_v \neq null \\
&\quad \wedge\, i'.i_v \neq null \\
&\quad \wedge\, ite\big(select(i'.i_v,\ i'.pos) = select(c'.c_v, i'.pos), \\
&\qquad\qquad select(i'.i_v,\ i'.pos) = v, \\
&\qquad\qquad select(i'.i_v,\ i'.pos) = select(c'.c_v,\ c'.size)\big)
\end{aligned} \tag{4.15}
$$

where v can not be an uninitialized object, c must be non-empty, i' must point to the found element or must point to $select(c'.c_v,\ c'.size)$, and $c'.c_v$ and $i'.i_v$ are equivalent.

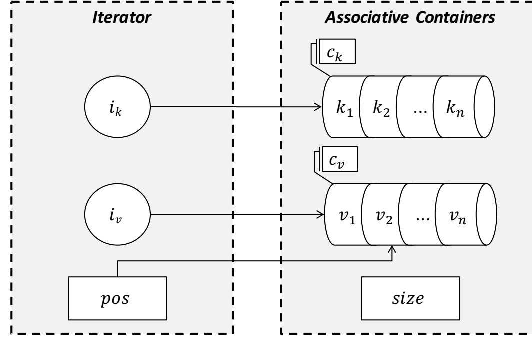


Figure 4.5: Operational model illustration of the STL associative containers.

4.2.3 Associative Containers

The associative container group consists of four classes: `map`, `multimap`, `set`, and `multiset`. The `map` class implements an associative container, where all elements are a combination of a key (i.e., $k \in K$) and a value (i.e., $v \in V$), where each value is associated with a unique key. On the one hand, elements are internally sorted by their keys based on a strict weak ordering rule [55]. On the other hand, `set` class presents a single-valued container that stores elements also following a strict weak ordering rule. Importantly, each value in a set must be unique, once the value is itself the key [55]. Finally, `multimap` and `multiset` classes behave similarly to `map` and `set` classes, respectively; however, both implement containers where multiple keys can be associated with multiple values. Fig. 4.5 gives an overview of our operational model for the STL associative containers.

In order to implement associative containers, the model consists of a pointer $c.c_v$, for the container's values, a pointer $c.c_k$, for the container's keys, and an integer $c.size$, for the container's size. Particularly, $c.c_k$ and $c.c_v$ are connected by an index, thus, an element in a given position n in $c.c_k$ (i.e., $select(c.c_k,\ n)$) is the key associated with the value in the same position n in $c.c_v$ (i.e., $select(c.c_v,\ n)$). Similarly, iterators for associative containers consist of a pointer $i.i_k$ that points to the same memory location as $c.c_k$, a pointer $i.i_v$ that points to the same memory location as $c.c_v$ and an integer $i.pos$ that indexes both $i.i_k$ and $i.i_v$. Once again, all operations in those libraries can be expressed as a simplified variation of the three main ones, i.e., insertion $(C.insert(K, V))$, deletion $(C.erase(I))$, and search $(C.search(K))$.

Firstly, the order of keys matters in the insertion operation for associative containers. Therefore, given a container c, the method call $c.insert(k, v)$ inserts the value v associated with the key k into the right order and it returns an iterator that points to the inserted position; however, if k already exists, the insertion is not performed and the method returns an iterator that points to the existing

element. Thus,

$$
\begin{aligned}
&\mathcal{C}((c', i') = c.insert(k, v)) := \\
&\quad ite(exists(k, c.c_k), \\
&\qquad \big(i'.pos = selectPosition(k, c.c_k) \\
&\qquad \wedge\; c'.c_k = c.c_k \\
&\qquad \wedge\; c'.c_v = c.c_v \\
&\qquad \wedge\; c'.size = c.size\big), \\
&\qquad \big(i'.pos = selectOrder(k, c.c_k) \\
&\qquad \wedge\; \forall n_2 \cdot (i'.pos \leq n_2 \leq c.size) \Rightarrow \\
&\qquad\qquad\qquad c'.c_k = store(c.c_k, n_2 + 1, \\
&\qquad\qquad\qquad\qquad select(store(c.c_k, n_2, k_2), n_2)) \\
&\qquad \wedge\; \forall n_3 \cdot (i'.pos \leq n_3 \leq c.size) \Rightarrow \\
&\qquad\qquad\qquad c'.c_v = store(c.c_v, n_3 + 1, \\
&\qquad\qquad\qquad\qquad select(store(c.c_v, n_3, v_2), n_3)) \\
&\qquad \wedge\; c'.c_k = store(c'.c_k,\, i'.pos,\, k) \\
&\qquad \wedge\; c'.c_v = store(c'.c_v,\, i'.pos,\, v) \\
&\qquad \wedge\; c'.size = c.size + 1\big), \\
&\quad \wedge\, i'.i_k = c'.c_k \\
&\quad \wedge\, i'.i_v = c'.c_v
\end{aligned}
\tag{4.16}
$$

note the side-effects $exists(k, c.c_k)$ that checks whether k exists in $c.c_k$, $selectPosition(k, c.c_k)$ that returns the position of k in $c.c_k$, and $selectOrder(k, c.c_k)$ that returns the position in which k must be inserted in $c.c_k$. The model in Eq. 4.16 induces the following properties,

$$
\begin{aligned}
&\mathcal{P}((c', i') = c.insert(k, v)) := \\
&\quad k \neq null \\
&\quad v \neq null \\
&\quad \wedge\, c.c_k \neq null \\
&\quad \wedge\, c.c_v \neq null \\
&\quad \wedge\, \forall n \cdot (0 \leq n \leq c.size) \Rightarrow \big(k \neq select(c.c_k, n)\big)
\end{aligned}
\tag{4.17}
$$

where we define as preconditions that v and k can not be uninitialized objects as well as k must be different from all keys within the c container (i.e., $\forall\, n \cdot (0 \leq n \leq c.size) \Rightarrow (k \neq select(c.c_k,\, n))$). Importantly, the comparison to check whether there exists already an element with the specified key is bypassed for containers that allow multiple keys.

Delete 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 4.2.2), the model for such operation

basically shifts backwards all elements followed by that specific position i. Thus,

$$\begin{aligned}
&\mathcal{C}((c', i') = c.erase(i)) := \\
&\quad \forall\, n_1 \cdot (i.pos + 1 \;\leq\; n_1 \;\leq\; c.size) \;\Rightarrow \\
&\qquad c'.c_k = store(c.c_k,\; n_1 + 1, \\
&\qquad\qquad select(store(c.c_k, n_1, k), n_1))\big) \\
&\quad \wedge \forall\, n_2 \cdot (i.pos + 1 \;\leq\; n_2 \;\leq\; c.size) \;\Rightarrow \\
&\qquad c'.c_v = store(c.c_v,\; n_2 + 1, \\
&\qquad\qquad select(store(c.c_v, n_2, v), n_2))\big) \\
&\quad \wedge c'.size = c.size - 1 \\
&\quad \wedge i'.i_k = c'{}_k \\
&\quad \wedge i'.i_v = c'{}_v \\
&\quad \wedge i'.pos = i.pos + 1
\end{aligned} \tag{4.18}$$

that implicitly induces the following properties,

$$\begin{aligned}
&\mathcal{P}((c', i') = c.erase(i)) := \\
&\quad i.i_k \;\neq\; null \\
&\quad i.i_v \;\neq\; null \\
&\quad \wedge\, 0 \;\leq\; i.pos \;\leq\; c.size \\
&\quad \wedge\, c.size \neq 0 \\
&\quad \wedge\, select(c'.c_k,\; i'.pos) \;=\; select(c.c_k,\; i.pos + 1) \\
&\quad \wedge\, select(c'.c_v,\; i'.pos) \;=\; select(c.c_v,\; i.pos + 1)
\end{aligned} \tag{4.19}$$

which are similar to the properties 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 modeled 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 respective element; otherwise, it returns an iterator that points to the position immediately after the last container's element (i.e., $c.end()$). Importantly, this search operation is implemented as a binary search algorithm [35], which implicitly produces the following properties,

$$\begin{aligned}
&\mathcal{P}((c', i') = c.search(k)) := \\
&\quad k \;\neq\; null \\
&\quad \wedge\, c.c_k \neq null \\
&\quad \wedge\, c.c_v \neq null \\
&\quad \wedge\, i'.i_v \neq null \\
&\quad \wedge\, i'.i_k \neq null \\
&\quad \wedge\, ite\big(select(i'.i_k,\; i'.pos) \;=\; select(c'.c_k, i'.pos), \\
&\qquad\quad select(i'.i_k,\; i'.pos) = k, \\
&\qquad\quad select(i'.i_k,\; i'.pos) = select(c'.c_k,\; c'.size)\big)
\end{aligned} \tag{4.20}$$

that are also similar to the properties held by the *search* method from sequential containers, except that $i'.i_k$ must point to the found element's key or i' must be equal to $c.end()$ as well as $c'.c_k$ and $i'.i_k$ must point to the same memory location.

4.2.4 Correctness

Our verification method depends on the fact that COM correctly represents the original SCL. Indeed, the correctness of such model to trust in the verification results is actually a major concern [50,68–70, 77,94,95]. In order to achieve that, the entire operational model was manually verified by checking the input/output relation through our test suite according to the original SCL specification [55] with the goal of ensuring the same behavior. Importantly, the operational model contains pre- and postconditions to guarantee that a (given) predicate holds before and after the execution of a (given) operation, respectively [70]. Although COM is a completely new implementation, it consists in (reliably) building a simplified model of the related SCL, using the C/C++ programming language through the ESBMC intrinsic functions (e.g., `assert` and `assume`) and the original documentation, which thus tend to reduce the number of programming errors. In addition, Cordeiro *et al.* [28,29] presented the soundness for such native 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 [63]. Conformance testing with respect to operational models would be a suitable approach [21, 70], but this option is not available in the present case; however, it would represent an exciting approach for future research.

4.3 Inheritance & Polymorphism

C++ features as inheritance and polymorphism make static analysis difficult to implement. In contrast to Java, which only allows single inheritance [34], where derived classes have only one base class, C++ also allows multiple inheritance, where a class may inherit from one or more unrelated base classes [35]. This particular feature makes C++ programs harder to model check than programs in other object-oriented programming languages (e.g., Java) since it disallows the direct transfer of techniques developed for other, simpler programming languages [2,76]. Indeed, multiple inheritance in C++ includes repeated and shared inheritance of base classes, object identity distinction, dynamic dispatch among other features that raise interesting challenges for model checking [81].

In C++, if a class inherits from a base class that does not contain virtual methods, then we call this *replicated inheritance*. If there exists a path from class X to class Y whose first edge is virtual, then we call this *shared inheritance*. 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 [55]. In particular, we follow these specifications to handle multiple inheritance, and to avoid issues such as name clashing when replicating the methods and attributes, e.g., 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 2.3).

A formal description to represent the relationship between classes can be described by a class

hierarchy graph (CHG). This graph is represented by a triple $\langle C, \prec_s, \prec_r \rangle$, where C is the set of classes, $\prec_s \subseteq C \times C$ refers to *shared inheritance* edges, and $\prec_r \subseteq C \times C$ are *replicated inheritance* edges. We also define the set of all inheritance edges $\prec_{sr} = \prec_s \cup \prec_r$. $(C, \leq_{sr})$ is then a partially ordered set [74] and $\leq_{sr}$ 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 [35]; 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, Fig. 4.6 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 $\langle C, \emptyset, \{(\texttt{JetCar}, \texttt{Car}), (\texttt{JetCar}, \texttt{Jet})\}\rangle$.

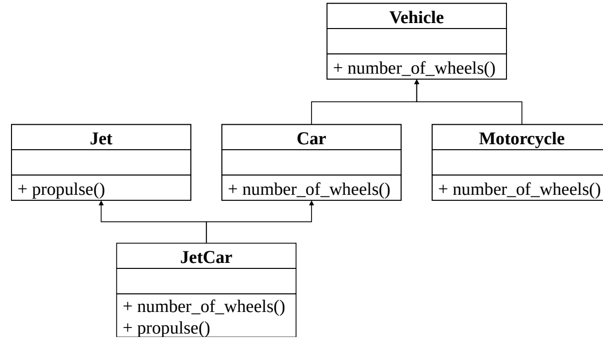


Figure 4.6: `Vehicle` class hierarchy UML diagram.

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. On the one hand, this approach has the advantage of having direct access to the attributes and methods of the derived class. It thus allows a more accessible validation, as the tool does not search for attributes or methods from base classes on each access. On the other hand, we replicate information to any new class, thus wasting memory resources.

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 Fig. 4.6, 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 [35]. In contrast

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

Figure 4.7: C++ program using a simplified version of the UML diagram in Fig. 4.6. The program nondeterministically cast derived class to a base class (either `Vehicle` or `Motorcycle` to `Car`). The goal is to check if the correct `number_of_wheels`() is called, from the base class.

to other approaches (e.g., the one proposed by Blanc, Groce, and Kroening [15]), 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.

In order to handle polymorphism, i.e., allowing variable instances to be bound to references of different types, related by inheritance [1], ESBMC implements a virtual table (i.e., `vtable`) mechanism [37]. When a class defines a virtual method, ESBMC creates a virtual table, which contains one pointer to each virtual method in the class. On the one hand, if a derived class does not override a virtual method, then the pointers are simply copied to the virtual table of the derived class. On the other hand, 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. Whenever a virtual method is called, ESBMC executes the method pointed in the virtual table.

Consider the program in Fig. 4.7, which contains a simplified version of the class hierarchy presented in Fig. 4.6. 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` in order to free the pointer `v` to simplify the GOTO instructions.

```
1  main() (c::main):
2     FUNCTION_CALL: return_value_nondet$1=nondet()
3     bool foo;
4     foo = return_value_nondet$1;
5
6     class Vehicle * v;
7     IF !foo THEN GOTO 1
8     new_value1 = new class Motorcycle;
9     new_value1->vtable->number_of_wheels =
10      &Vehicle::number_of_wheel();
11    new_value1->vtable->number_of_wheels =
12      &Motorcycle::number_of_wheel();
13    v = (class Vehicle *)new_value;
14    GOTO 2
15 1: new_value2 = new class Car;
16    new_value2->vtable->number_of_wheels =
17      &Vehicle::number_of_wheel();
18    new_value2->vtable->number_of_wheels =
19      &Car::number_of_wheel();
20    v = (class Vehicle *)new_value;
21    bool res;
22 2: IF !foo THEN GOTO 3
23    FUNCTION_CALL: return_value_number_of_wheels =
24      *v->vtable->number_of_wheel()
25    res = wheels == 2
26    GOTO 4
27 3: FUNCTION_CALL: return_value_number_of_wheels =
28      *v->vtable->number_of_wheel()
29    res = wheels == 4
30 4: ASERT res
31    RETURN: 0
32 END_FUNCTION
```

Figure 4.8: GOTO instructions generated for the program in Fig. 4.7.

Fig. 4.8 shows the GOTO program generated for the program in Fig. 4.7. Note that, when building the polymorphic object `v`, the virtual table pointer for the method `number_of_wheel`() in the GOTO program is first assigned with a pointer to the method `number_of_wheel`() in class `Vehicle` (see lines 10 and 17 in Fig. 4.8); this happens because the constructor for both `Car` and `Motorcycle` first call the base constructor in the original program (see lines 11 and 18 in Fig. 4.7). They are then assigned the correct method address (see lines 12 and 19 in Fig. 4.8) in the constructors of the derived classes, i.e., `Motorcycle` and `Car`, respectively.

In the SSA form shown in Fig. 4.9, every branch creates a separate variable, which are then

```
1  return_value_nondet1 = nondet_symbol(symex::0)
2  foo1 = return_value_nondet1
3  new_value11 = new_value10
4    WITH [vtable = new_value10.vtable
5      WITH [number_of_wheel =
6        &Motorcycle::number_of_wheels()]]
7  new_value12 = new_value11
8    WITH [vtable = new_value11.vtable
9      WITH [number_of_wheel =
10       &Motorcycle::number_of_wheels()]]
11 v1 = new_value12
12 new_value21 = new_value20
13   WITH [vtable = new_value20.vtable
14     WITH [number_of_wheel =
15       &Motorcycle::number_of_wheels()]]
16 new_value22 = new_value21
17   WITH [vtable = new_value21.vtable
18     WITH [number_of_wheel =
19       &Motorcycle::number_of_wheels()]]
20 v2 = new_value22
21 v3 = (foo1 ? v1 : v2);
22 return_value_number_of_wheels1 = 2
23 res1 = (return_value_number_of_wheels1 == 2)
24 return_value_number_of_wheels2 = 4
25 res2 = (return_value_number_of_wheels2 == 4)
26 res3 = (foo1 ? res1 : res2)
```

Figure 4.9: The program of Fig. 4.7 in SSA form.

merged when the control-flow merges. Here, we generate two branches (i.e., `v1` and `v2`) and a ϕ-node (i.e., `v3`) to merge both 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 21 in Fig. 4.9). There exists no side-effect in the SSA form, as the result of the `number_of_wheels`() is propagated. Note that the asserts are not presented in the SSA form, but will be converted into SMT formulae. ESBMC builds the constraints and properties, as shown in Eqs. (4.21) and (4.22), respectively, based on the SSA and the assertions.

$$\mathcal{C} := \left[\begin{array}{l} return_value_number_of_wheels_1 = 2 \\ \wedge\ res_1 = (return_value_number_of_wheels_1 = 2) \\ \wedge\ return_value_number_of_wheels_2 = 4 \\ \wedge\ res_2 = (return_value_number_of_wheels_2 = 4) \\ \wedge\ res_3 = ite\,(foo1, res_1, res_2) \end{array}\right] \tag{4.21}$$

$$\mathcal{P} := \left[\ res_3 = 1\ \right] \tag{4.22}$$

4.4 Exception Handling

Exceptions are unexpected circumstances that arise during the execution of a program, e.g., runtime errors [35]. 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. Fig. 4.10 shows an example of C++ program where an exception is thrown inside a try block.

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

Figure 4.10: Try-catch example of throwing an integer exception.

In order 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, while in the latter we implemented the rules for throwing and catching an exception, as well as the control flows for the unexpected and terminate handlers.

The GOTO conversion slightly modifies the exception handling blocks: a try block is represented using several instructions: 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 begin and end of the try block, however, they 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 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. Fig. 4.11 shows the GOTO instructions generated from the code shown in Fig. 4.10.

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 Section 4.4, including a jump to an invalid catch that triggers a verification error, i.e., it represents an exception thrown that can not 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

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

Figure 4.11: Try-catch conversion to GOTO instructions.

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, and 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, e.g., if an object is allocated inside a try block, and an exception is thrown and handled, it will leak unless the reference to the allocated memory is somehow tracked and freed.

Our symbolic engine also keeps track of *function frames*, i.e., several pieces of information about the function it is currently evaluating, including arguments, recursion depth, local variables and others. These pieces of information are not only important because we want to handle recursion or find memory leaks, but also allows us to connect exception thrown outside the scope of a function, and to handle exception specification [42].

```
1 return_value_nondet1 = nondet_symbol(symex::0)
2 i1 = 20
3 f1 = 10.0f
4 f2 = (return_value_nondet1 ? f0 : f1)
5 i2 = (return_value_nondet1 ? i1 : i0)
```

Figure 4.12: The program of Fig. 4.10 in SSA form.

As an illustrative example, for the program in Fig. 4.10, ESBMC generates the SSAs shown in Fig. 4.12. The SSA form only contains conditional and unconditional assignments. Note the use of free variables f_0 and i_0, and they are used to model paths where no values are assigned to f or i (in this case, however, these paths are never reachable). ESBMC then builds the constraints and properties as shown in Eqs. (4.23) and (4.24), and the formula $\mathcal{C} \wedge \neg\mathcal{P}$ is given to an SMT solver to check for satisfiability.

$$\mathcal{C} := \left[\begin{array}{l} i_1 = 20 \wedge f_1 = 10.0f \\ \wedge\ f_2 = ite\,(return_value_nondet1, f_0, f_1) \\ \wedge\ i_2 = ite\,(return_value_nondet1, i_1, i_0) \end{array} \right] \tag{4.23}$$

$$\mathcal{P} := \begin{bmatrix} i_2 = 20 \\ \wedge\ f_2 = 10.0f \end{bmatrix}, \tag{4.24}$$

where Eq. (4.24) is always evaluated to false, i.e., the program is safe. Note that since the program can not be statically determined because of the nondeterministic call (line 4 in Fig. 4.10), the program is symbolic encoded and all jumps to catch statements are encoded.

A C++ program can throw an exception in several situations other than the usage of explicit `throw` expression (e.g., an exception of type `bad_alloc` can be thrown by operator `new`, an exception of the type `bad_cast` can be thrown by the `dynamic_cast` operator or even the function `typeid` can throw an exception of type `bad_typeid`). Those exceptions are built into the C++ language and are supposed to be handled by the program. In order to properly define the verification of exception handling in C++, we formally define two syntactic domains, including exceptions E and handlers H as follows:

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

In this context, e and h are classes of variables of type E and H, respectively. We abuse 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 abuse 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*`), $h_{\dots}$ is a `catch` statement of type ellipsis [55], 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 to the type of a given handler h as shown in Eq. (4.25). Furthermore, we declare the 1-arity function $\zeta : H^* \longmapsto H$ 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 [55] for the type of a thrown exception e as shown in Eq. (4.26). 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 [55] as shown in Eq. (4.27).

$$M(e,h) \stackrel{\text{def}}{=} \begin{cases} \top, & \text{type of } e \text{ is matches to the type of } h \\ \bot, & \text{otherwise} \end{cases} \tag{4.25}$$

$$U(e,h) \stackrel{\text{def}}{=} \begin{cases} \top, & c \text{ is an unambiguous base of } e \\ \bot, & \text{otherwise} \end{cases} \tag{4.26}$$

$$Q(e,h) \stackrel{\text{def}}{=} \begin{cases} \top, & e \text{ can be implicit converted to } h \\ \bot, & \text{otherwise} \end{cases} \tag{4.27}$$

The C++ language standard defines rules to connect `throw` expressions and `catch` statements [55], which are all described in Table 4.2. Each rule represents a function $r_k : E \longmapsto H$ 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 Eq. (4.28), after the evaluation of all rules (i.e., $h_{r_1}, ..., h_{r_9}$), ESBMC returns the first handler h_{r_k} that matched the thrown exception e.

$$\begin{aligned}
\mathcal{H}(e, h_1, ..., h_n) :=& \\
& h_{r_1} = r_1(e, h_1, ..., h_n) \\
& \wedge \ldots \\
& \wedge\ h_{r_9} = r_9(e, h_1, ..., h_n) \\
& \wedge\ ite(h_{r_1} \neq h_{null}, h_{r_1}, \\
& \quad\ ite(h_{r_2} \neq h_{null}, h_{r_2}, \\
& \quad\ \ldots \\
& \quad\ ite(h_{r_9} \neq h_{null}, h_{r_9}, h_{null}) \ldots)
\end{aligned} \tag{4.28}$$

Table 4.2: Rules to connect `throw` expressions and `catch` blocks.

Rule	Behavior	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 [55].	$ite(\exists h \cdot\ e = e_* \wedge h = h_* \wedge Q(e_*, h_*), h_{r_6} = h_*, h_{r_6} = h_{null})$
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.	$ite(e = e_{null} \wedge e_{-1} \neq e_{null},$ $h'_{r_1} = r_1(e_{-1}, h_1, ..., h_n)$ $\wedge\ \ldots$ $\wedge\ h'_{r_9} = r_9(e_{-1}, h_1, ..., h_n),$ $h_{r_9} = h_{null})$

Chapter 5

Experimental Evaluation

The experimental evaluation compares ESBMC against LLBMC and DIVINE regarding correctness and performance in the verification process of C++ programs; DIVINE was developed by Baranová et al. [4], and LLBMC was developed by Merz, Falke, and Sinz [40]. Section 5.2 shows a detailed description of all tools, scripts, and benchmark dataset, while Section 5.4 presents the results and our evaluation. Our experiments are based on a set of publicly available benchmarks. All tools, scripts, benchmarks, and the results of our evaluation are available on a supplementary web page at `http://esbmc.org/`.

5.1 Objectives

Experiments aim at answering two questions regarding *correctness* and *performance* of ESBMC:

(EQ-I) How accurate is ESBMC when verifying the chosen C++03 programs?

(EQ-II) How does ESBMC performance compare to other existing model checkers?

To answer both questions, we evaluate all benchmarks with ESBMC v2.0, DIVINE v4.0.22, and LLBMC v2013.1. Our experimental evaluation does not include CBMC v5.8 [24], because its C++ support is still very rudimentary and it fails during the verification of 99% of all benchmarks (mostly during the parser step), as already reported by Merz et al. [40], Monteiro et al. [70], and Ramalho et al. [80].

5.2 Description of the Benchmarks

To tackle modern aspects of the C++ language, the comparison is based on a benchmark dataset that consists of 1,513 C++03 programs. In particular, 290 programs were extracted from Deitel

& Deitel [35], 432 were extracted from C++ Resources Network [19], 16 were extracted from NEC Corporation [98], 16 programs were obtained from LLBMC [40], 39 programs were obtained from CBMC [24], 55 programs were obtained from the GCC test suite [41], and the others were developed to check several features of the C++ programming language [80]. 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[1] and CBMC[2] test suite, which are specific to templates.

5.3 Experimental Setup

All experiments were conducted on a computer with an i7-4790 processor, 3.60GHz clock, with 16GB RAM memory and Ubuntu 14.04 64-bit OS. ESBMC, LLBMC, and DIVINE were set to a time limit of 900 seconds (i.e., 15 minutes) and a memory limit of 14GB. All presented execution times are actually CPU times, i.e., only the elapsed time 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,[3] 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,[4] [59] then it reads the parameters from a file and executes the tool;[5] and the last one for DIVINE, which also first pre-compiles the C++ program to bitcode, then performs the verification on it[6]. All parameters were based on previous publications and confirmed by developers. 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

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

(i.e., with the $-\texttt{fno}-\texttt{exceptions}$ flag of the compiler). If exception handling is enabled, then LLBMC always aborts the verification process.

5.4 Experimental Results

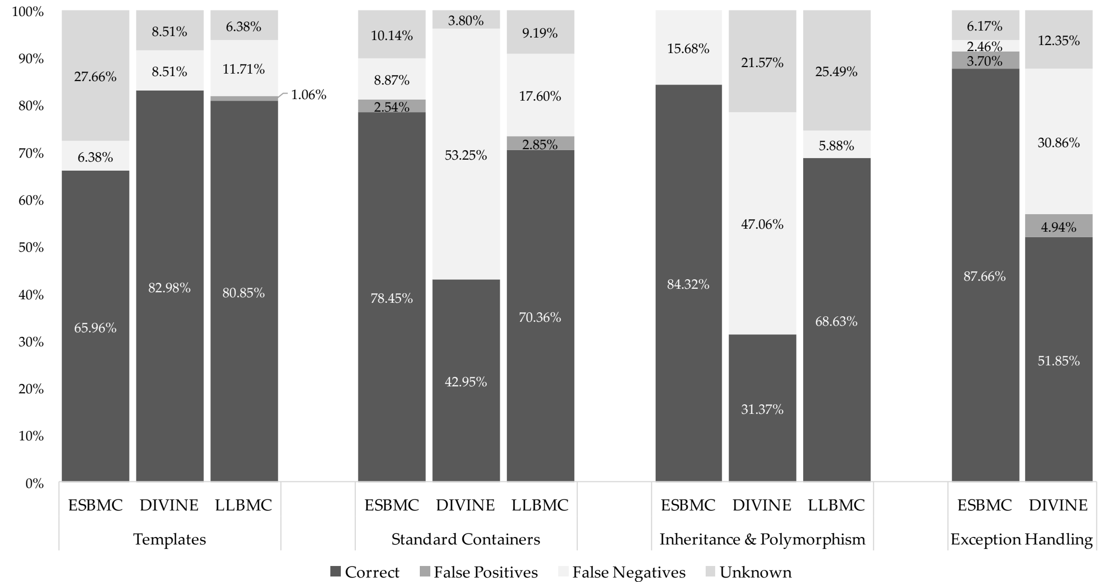


Figure 5.1: Comparison of verification results between ESBMC $v2.0$, DIVINE $v4.0.22$, and LLBMC $v2013.1$.

In this section, we present the results using percentages (in relation to the 1,513 C++ benchmarks), as shown in Fig. 5.1. The data used to generate these percentages is available at the Appendix A. *Correct* represents the positive results, *i.e.*, 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 is correct. Finally, *Unknown* represents the benchmarks where each tool aborted the verification process due to internal errors, timeout (i.e., the tool was killed after 900 seconds) or 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 always aborts the verification process. Furthermore, to better present the results of our experimental evaluation, the test suites were grouped into five categories:

Templates – formed by the *cbmc*, *gcc-templates* and *templates* test suites (94 benchmarks);

Standard Containers – formed by *algorithm*, *deque*, *vector*, *list*, *queue*, *priority_queue*, *stack*, *map*, *multimap*, *set* and *multiset* test suites (631 benchmarks);

Inheritance & 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*, and *stream* test suites (656 benchmarks).

On the Templates category (see Fig. 5.1), DIVINE presented the best results and reached a successful verification rate of 82.98%, while LLBMC reported 79.79% and ESBMC 65.96%. DIVINE does not report any timeout, memory out, or false-positive results for this category, but it reports false-negative results in 8.51% of the benchmarks due to false claims of uncaught exceptions, and a bug related to the naming convention adopted by LLVM for the macro `__PRETTY_FUNCTION__` in template methods. It also reports unknown in 8.51% of the benchmarks due to either not supporting friends template declarations [55] or due to errors when verifying template specialization. LLBMC reports a false-negative rate of 11.71% and a false-positive rate of 1.06%, which are related to problems with invalid data members from nested templates or specialized template classes, handling parameters of template functions, and errors with operator overloading. Furthermore, LLBMC presents problems to deal with template specialization that leads to an unknown rate of 6.38%. ESBMC does not report any timeout, memory out, or false-positive results in this category, but a false-negative rate of 6.38% and an unknown rate of 27.66%, both due to problems instantiating nested templates, where the tool wrongly chooses the instantiated template to be used during the verification or the lack of support for friends template declarations (cf., Section 4.1).

Although the current support of templates was sufficient to verify real-world C++ applications, as shown in Fig. 5.2, it is still work-in-progress. For instance, the handling of SFINAE [55] in ESBMC is limited, and limitations on the support of nested templates, as shown in the experiments, directly affect the verification process. Furthermore, our custom front-end for C++ in ESBMC offers support only to C++03 templates; template features from newer standards (e.g., variadic templates) are not supported. This limitation is due to the fact that template instantiation is notoriously hard, especially if we consider more recent standards and, although our front-end can handle a number of real-world C++ programs, maintaining the C++ front-end in ESBMC is a herculean task. We plan to develop a new C++ front-end based on clang's AST, similar to the C front-end already in place (see Section 6).

On the Standard Containers category (see Fig. 5.1), ESBMC presented the best results and reached a successful verification rate of 78.45%, while LLBMC reported 70.36% and DIVINE 42.95%. ESBMC's noticeable results for containers are directly related to its COM. The majority of the benchmarks for this category contain assertions checking functional aspects of the containers and its operations, e.g., to check whether the `operator`[] from a `vector` object is called with an argument out of range, which would lead to undefined behavior (*cf.*, Section 4.2). In this context, pre- and

postconditions embedded into COM extend the capabilities of ESBMC to not only check for memory-safety properties (*e.g.*, null pointer dereferences or arithmetic overflows), but also to reason about the aforementioned functional aspects of the containers. 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; we are currently working to address them in future versions. ESBMC also reported 10.14% of unknown results due to the aforementioned template limitations. LLBMC reports a false-positive rate of 2.85% and a false-negative rate of 17.60%, which is mostly related to functional properties (e.g., assertions to check whether the 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 [40]. DIVINE does not report any timeout, memory out, or false-positive results for this category, but an expressive false-negative rate of 53.25%, which is a result of errors to check assertions regarding functional properties (similar to LLBMC). DIVINE also reports an unknown rate of 3.80% due to errors with pointer handling, probably due to imprecise (internal) encoding, or errors to check assertions regarding functional properties.

On the Inheritance & Polymorphism category (see Fig. 5.1), ESBMC presented the best results and reached a successful verification rate of 84.32% while LLBMC reported 68.63% and DIVINE 31.37%. ESBMC does not report any timeout or memory out, but it reports a false-negative rate of 15.68%, which is 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 an expressive false-negative rate of 47.06% 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 Fig. 5.1), ESBMC presented the best results and reached a successful verification rate of 87.66% while DIVINE reported 51.85%. 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 4.2 in ESBMC, i.e., "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, but it reports a false-positive rate of 4.94% and an expressive false-negative rate of 30.86%, where it incorrectly handles re-throws, exception specification and the unexpected as well as terminate function handlers. DIVINE also presents an unknown rate of 12.35% due to errors when dealing with exceptions thrown by derived classes instantiated as base classes.

5.5 Discussion

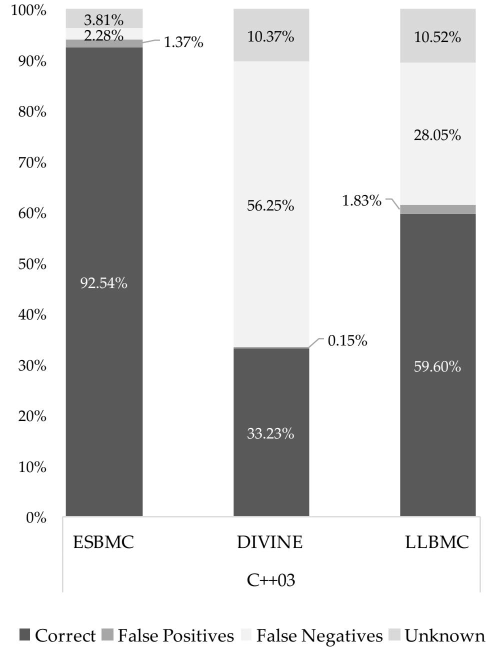


Figure 5.2: Comparison of verification results between ESBMC $v2.0$, DIVINE $v4.0.22$, and LLBMC $v2013.1$ regarding general-purpose benchmarks in C++03.

To evaluate how these model checkers perform when applied to general C++03 benchmarks, we evaluate them against the category C++03. The category covers programs up to the 2013 standard because ESBMC front-end does not fully support newer versions yet [80]. In this category, model checkers deal with benchmarks that not only make use of an specific feature of the C++ language (e.g., exception handling and containers), but how they deal with programs that make use of multiple object-oriented features (e.g., inheritance and exceptions), that contain a wider range of libraries from the STL, or manipulation of strings and streams, among other C++03 features. ESBMC presented the highest successful verification rate, 92.54%, followed by LLBMC 59.60% and DIVINE 33.23%. The successful expressive rate of ESBMC in this category not only correlate 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 4.1. 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 makes

use of at least one standard template library. ESBMC does not report any memory out, but it reports a false-positive rate of 1.37%, a false-negative rate of 2.28%, and an unknown rate of 3.81%, which are all due to the same issues pointed by the previous experiments. LLBMC reports a false-positive rate of 1.83% and a false-negative rate of 28.05%, 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 10.52% mostly regarding errors with operator overloading and the ones mentioned in the previous categories. DIVINE does not report any timeout or memory out, but an expressive false-negative rate of 56.25%, which is a result of errors when checking assertions that represent functional properties of objects across all STL (similar to LLBMC). DIVINE also reports one false positive regarding the instantiation of function template specialization, and an unknown rate of 10.37% due to crashes when handling pointers.

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. [9]; 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.

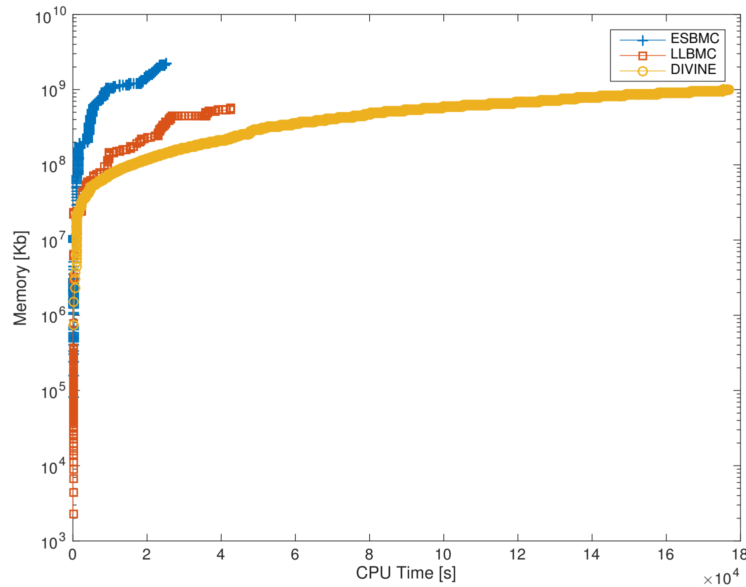


Figure 5.3: Comparison of accumulative verification time and accumulative memory consumption among ESBMC *v*2.0, DIVINE *v*4.0.22, and LLBMC *v*2013.1 throughout the verification process of all benchmarks.

Fig. 5.3 shows the accumulative verification time and accumulative memory consumption for the tools under evaluation. All the tools take more time to verify the test suites *algorithm*, *string*, and *cpp*, due to the large number of test cases and the presence of pointers and iterators. ESBMC is the fastest of the three tools, being 1.7 times faster than LLBMC and 7 times faster than DIVINE. In terms of verification time, DIVINE is the only tool that did not use more than the defined limit of 900 seconds, while ESBMC and LLBMC aborted due to timeout in 7 and 39 benchmarks, respectively. In terms of memory consumption, DIVINE is the only tool that did not use more than the defined limit of 14GB per benchmark, while ESBMC and LLBMC aborted due to exhaustion of the memory resources in 3 and 17 of them, respectively. Even so, LLBMC consumes less memory overall (i.e., 571.71GB) when compared to DIVINE (i.e., $1,017.19$GB) and ESBMC (i.e., $2,204.11$GB).

Based on the entire experimental set (see Appendix A), ESBMC achieved the highest success rate of 84.27% in 25251 seconds (approximately 7 hours), faster than both LLBMC and DIVINE, which positively answers our experimental questions EQ-I and EQ-II. LLBMC correctly verified 62.52% in 44438 seconds (approximately 12.34 hours) and is only able to verify the programs if exception handling is disabled, which is not a problem for both ESBMC and DIVINE. DIVINE correctly verified 41.31% in 176957 seconds (approximately 49.15 hours). Regarding memory usage, ESBMC has the highest usage among the three tools, which is 2.16 and 3.85 times higher than DIVINE and LLBMC, respectively; this high consumption is due to the generation process of SSA forms (cf., Sec. 4); however, its optimization is under development and will be available in future versions. 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, but the current work-in-progress clang frontend 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 the application of model checking in C++ programs.

Chapter 6

Conclusions

This work presented a novel SMT-based BMC approach to verify C++ programs using ESBMC. Firstly, we describe our approach to support templates, which works similarly as conventional compilers, by replacing the instantiated templates before the encoding phase. We also present an abstraction of the standard C++ libraries, named COM, which replaces the SCL during the verification of a C++ program and whose purpose is to verify safety properties related to the usage of the SCL. Furthermore, we describe ESBMC's mechanism to handle single and multiple inheritance and polymorphism in C++ programs. Lastly, we formally describe and all throw & catch exception rules in ESBMC. The main contributions of this work are the formalization of the ESBMC's engine to handle templates, inheritance & polymorphism, and exception handling; the operational model structure to handle new features from the SCL (e.g., sequential and associative template-based containers); and the expressive set of publicly available benchmarks explicitly designed to evaluate model checkers that target the C++ programming language.

In order to evaluate the proposed approach, we introduce a new version of ESBMC $v2.0$ with a new set of over $1,500$ benchmarks, which covers several features offered by the C++03 programming language. It is worth noting that ESBMC can verify correctly 84.27%, in approximately 7 hours, outperforming other state-of-art C++ verification tools (cf., Chapter 5). In the experimental evaluation, two state-of-art verifiers, DIVINE and LLBMC, are compared to ESBMC, which was able to present a higher number of correct results, in less time than both tools (7 and 1.7 times faster than DIVINE and LLBMC, respectively). Importantly, ESBMC and DIVINE were also able to verify programs with exceptions enabled, a missing feature from LLBMC that decreases the verification accuracy of real-world C++ programs.

Based on the extensive experimental evaluation and the available literature, we believe this work sets ESBMC as the state-of-the-art in the verification of C++ programs regarding inheritance & polymorphism, standard C++ libraries, and exception handling.

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 intend to develop an automatic conformance testing procedure to ensure that our COM conservatively approximates the SCL semantics.

Furthermore, we intend to improve the general verification of C++ programs, including improved support for templates. Although our C++ front-end is able to support most features of C++03, to improve the front-end for newer versions of the C++ standard is unmanageable. For that reason, we could rewrite our front-end using clang [59] to generate the program AST for C++ programs. To obtain full control over the clang AST and integrate it into ESBMC's front-end, we could use the LibTooling[1] API, which is a C++ interface aimed at writing standalone tools based on clang. Importantly, we do not intend to use the LLVM intermediate representation, but the AST generated by clang. An AST is a representation of the abstract syntactic structure of source code, which does not represent every aspect in the syntax, but rather just the specific language elements that have meaning [49]. In particular, if we use clang to generate the AST, then it solves several problems:

- 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.
- Supporting new features should be as easy as adding a new AST conversion node from the clang representation to ESBMC representation.
- We do not need to maintain a full C++ front-end since ESBMC would 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 [44], and the C++ front-end is currently under development. One of the major bottlenecks to maintain the ESBMC support of the C++ programming language is the need to update its front-end for each C++ release. Rewrite the ESBMC's front-end for C++ programs requires a significant engineering effort. We already started the development of the new C++ front-end. It is, however, far from complete. Therefore, we want to focus on the object-oriented aspects to set the foundation of this approach. Firstly, we intend to extend and evaluate ESBMC to verify the fundamental structures of object-oriented programs (*e.g.*, classes, methods, constructors, and destructors).

Furthermore, we must improve and evaluate ESBMC to include the instantiation of all templates in the program, which represents the weakest aspect in the current ESBMC version, as shown in the experimental evaluation (cf. Chapter 5). Finally, we would need to focus again on the support

[1] http://clang.llvm.org/docs/LibTooling.html

for inheritance & polymorphism, including the code to build virtual tables. Unfortunately, the AST provided by clang does not represent the whole type-structure of C++, virtual classes and virtual tables are present. There are cases where the front-end can deduce the type of an object, but it is not always so. There are two possible solutions here: we could either implement a full virtual dispatch table or use some devirtualization approach to solving the problem. The idea is to use static analysis to restrict/resolve the virtual types, the use of dispatch tables only when necessary. This work would set a strong foundation for the full support of the C++ programming language in ESBMC.

Bibliography

[1] Roger T. Alexander, Jeff Offutt, and James M. Bieman. Fault detection capabilities of coupling-based OO testing. In *Software Reliability Engineering*, pages 207–2002, 2002.

[2] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. JPF-SE: A symbolic execution extension to java pathfinder. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 4424 of *LNCS*, pages 134–138, 2007.

[3] Andrei Arusoaie, Stefan Ciobâca, Vlad Craciun, Dragos Gavrilut, and Dorel Lucanu. A comparison of open-source static analysis tools for vulnerability detection in C/C++ code. In *Symposium On Symbolic And Numeric Algorithms For Scientific Computing*, pages 161–168, 2017.

[4] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. Model checking of C and C++ with DIVINE 4. In *Automated Technology For Verification And Analysis*, volume 10482 of *LNCS*, pages 201–207, 2017.

[5] Clark Barrett, Christopher Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer-Aided Verification*, volume 6806 of *LNCS*, pages 171–177, 2011.

[6] Dirk Beyer. Competition on software verification (SV-COMP). In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 7214 of *LNCS*, pages 504–524, 2012.

[7] Dirk Beyer. Second competition on software verification - (summary of SV-COMP 2013). In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 7795 of *LNCS*, pages 594–609, 2013.

[8] Dirk Beyer. Status report on software verification - (competition summary SV-COMP 2014). In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 8413 of *LNCS*, pages 373–388, 2014.

[9] Dirk Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 9035 of *LNCS*, pages 401–416, 2015.

[10] Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 9636 of *LNCS*, pages 887–904, 2016.

[11] Dirk Beyer. Software verification with validation of results (report on SV-COMP 2017). In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 10206 of *LNCS*, pages 331–349, 2017.

[12] Dirk Beyer. Automatic verification of C and java programs: SV-COMP 2019. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 11429 of *LNCS*, pages 133–155, 2019.

[13] Dirk Beyer and Thomas Lemberger. Software verification: Testing vs. model checking. In *Hardware and Software: Verification and Testing*, volume 10629 of *LNCS*, pages 99–114, 2017.

[14] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*, volume 185. IOS Press, 2009.

[15] Nicolas Blanc, Alex Groce, and Daniel Kroening. Verifying C++ with STL containers via predicate abstraction. In *Automated Software Engineering*, pages 521–524, 2007.

[16] Aaron R. Bradley and Zohar Manna. *The Calculus Of Computation - Decision Procedures With Applications To Verification*. Springer, 1st edition, 2007.

[17] Aaron R. Bradley and Zohar Manna. *The Calculus Of Computation: Decision Procedures With Applications To Verification*. Springer-Verlag New York, Inc., 1st edition, 2007.

[18] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 5505 of *LNCS*, pages 174–177, 2009.

[19] C++ Resources Network. *Reference Of The C++ Language Library*. Standard C++ Foundation, 2013. [Online; accessed September-2018].

[20] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium On Operating Systems Design And Implementation*, pages 209–224, 2008.

[21] Pedro de la Cámara, J. Raúl Castro, María-del-Mar Gallardo, and Pedro Merino. Verification support for ARINC-653-based avionics software. *Software Testing, Verification and Reliability*, 21(4):267–298, 2011.

[22] Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Code-level model checking in the software development workflow. In *42nd International Conference on Software Engineering (ICSE)*, 2020.

[23] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The mathSAT5 SMT solver. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 7795 of *LNCS*, pages 93–107, 2013.

[24] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 2988 of *LNCS*, pages 168–176, 2004.

[25] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 3440 of *LNCS*, pages 570–574, 2005.

[26] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. *Handbook Of Model Checking*, chapter Introduction To Model Checking, pages 1–26. Springer International Publishing, 2018.

[27] Byron Cook, Kareem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Model checking boot code from AWS data centers. In *Computer Aided Verification*, pages 467–486, 2018.

[28] Lucas C. Cordeiro and Bernd Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. In *International Conference on Software Engineering*, pages 331–340, 2011.

[29] Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering*, 38(4):957–974, 2012.

[30] Lucas C. Cordeiro, Bernd Fischer, and Joã£o Marques-Silva. Continuous verification of large embedded software using SMT-based bounded model checking. In *Engineering of Computer Based System*, pages 160–169, 2010.

[31] Erickson H. da S. Alves, Lucas C. Cordeiro, and Eddie Batista de Lima Filho. Fault localization in multi-threaded C programs using bounded model checking. In *2015 Brazilian Symposium*

on Computing Systems Engineering, SBESC 2015, Foz do Iguacu, Brazil, November 3-6, 2015, pages 96–101. IEEE Computer Society, 2015.

[32] Erickson H. da S. Alves, Lucas C. Cordeiro, and Eddie Batista de Lima Filho. A method to localize faults in concurrent C programs. *J. Syst. Softw.*, 132:336–352, 2017.

[33] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 4963 of *LNCS*, pages 337–340, 2008.

[34] Harvey M. Deitel and Paul J. Deitel. *Java How To Program*. Prentice-Hall, Inc., 6th edition, 2004.

[35] Harvey M. Deitel and Paul J. Deitel. *C++ How To Program*. Prentice Hall Press, 6th edition, 2007.

[36] Dino Distefano, Manuel Fahndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at Facebook. *Communications of ACM*, 62:62–70, 2019.

[37] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in C++. In *Object-Oriented Programming, Systems, Languages & Applications*, pages 306–323, 1996.

[38] Bruno Dutertre. Yices 2.2. In *Computer-Aided Verification*, volume 8559 of *LNCS*, pages 737–744, 2014.

[39] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1st edition, 1990.

[40] Stephan Falke, Florian Merz, and Carsten Sinz. The bounded model checker LLBMC. In *Automated Software Engineering*, pages 706–709, 2013.

[41] Free Software Foundation, Inc. *GCC, The GNU Compiler Collection*. The LLVM Project, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1335 USA, 2015.

[42] Mikhail R. Gadelha. Verificação Baseada em Indução Matemática para Programas C++. Master's thesis, Federal University of Amazonas, Brazil, 2013.

[43] Mikhail R. Gadelha, Rafael Menezes, Felipe R. Monteiro, Lucas Cordeiro, and Denis Nicole. Esbmc: Scalable and precise test generation based on the floating-point theory (competition contribution). In *International Conference on Fundamental Approaches to Software Engineering*, volume 12076 of *LNCS*, pages 525–529, 2020.

[44] Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. ESBMC 5.0: An industrial-strength C model checker. In *Automated Software Engineering*, pages 888–891, 2018.

[45] Mikhail R. Gadelha, Felipe R. Monteiro, Lucas Cordeiro, and Denis 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*, volume 11429 of *LNCS*, pages 209–213, 2019.

[46] Mikhail Y. R. Gadelha, Lucas C. Cordeiro, and Denis A. Nicole. Encoding floating-point numbers using the SMT theory in ESBMC: an empirical evaluation over the SV-COMP benchmarks. In Simone André da Costa Cavalheiro and José Luiz Fiadeiro, editors, *Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings*, volume 10623 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2017.

[47] Mikhail Y. R. Gadelha, Hussama Ibrahim Ismail, and Lucas C. Cordeiro. Handling loops in bounded model checking of C programs via k-induction. *STTT*, 19(1):97–114, 2017.

[48] Mikhail Y. R. Gadelha, Felipe R. Monteiro, Lucas C. Cordeiro, and Denis A. Nicole. Towards counterexample-guided k-induction for fast bug detection. In *ACM Joint European Software Engineering Conference And The Foundations Of Software Engineering*, pages 765–769, 2018.

[49] D. Galles. *Modern Compiler Design*, chapter Abstract Syntax Trees in C, pages 33–48. Scott/Jones, 2004.

[50] Mário Garcia, Felipe R. Monteiro, Lucas C. Cordeiro, and Eddie Batista de Lima Filho. ESBMC^{QtOM}: A bounded model checking tool to verify qt applications. In *SPIN*, volume 9641 of *LNCS*, pages 97–103, 2016.

[51] Raphael Geissert. Debian automated code analysis. `https://qa.debian.org/daca/`, 2019. [Online; accessed August-2019].

[52] Benny Godlin and Ofer Strichman. Regression verification: Proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, 23(3):241–258, 2013.

[53] C. Hathhorn and G. Rosu. Dealing with C's original sin. *IEEE Software*, 36:24–28, 2019.

[54] Steven Holzner. *C++ Black Book*. Coriolis, 1st edition, 2000.

[55] ISO. *C++ Standard*, 2003. ISO/IEC 14882:2003.

[56] Franco Ivančić, Ilya Shlyakhter, Aarti Gupta, and Malay K. Ganai. Model checking C programs using F-SOFT. *Computer Design*, pages 297–308, 2005.

[57] Wei Ming Khoo, Saad Aloteibi, Ross Anderson, and Michael Meeks. Hunting for vulnerabilities in large software: The openoffice suite. *Cambridge University press*, page 9, 2010.

[58] Daniel Kroening, Joël Ouaknine, Ofer Strichman, Thomas Wahl, and James Worrell. Linear completeness thresholds for bounded model checking. In *Computer-Aided Verification*, volume 6806 of *LNCS*, pages 557–572, 2011.

[59] Chris Lattner. *Clang Documentation*. The Clang-LLVM Project, 2015. [Online; accessed September-2018].

[60] Bruno Cardoso Lopes and Rafael Auler. *Getting Started With LLVM Core Libraries*. Packt Publishing, 1st edition, 2014.

[61] Daniel Marjamäki. Cppcheck - a tool for static C/C++ code analysis. `http://cppcheck.sourceforge.net`, 2018. [Online; accessed August-2019].

[62] John McCarthy. *Program Verification: Fundamental Issues In Computer Science*, chapter Towards a Mathematical Science of Computation, pages 35–56. Springer Netherlands, 1993.

[63] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199(1):200–227, 2005.

[64] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *Verified Software: Theories, Tools, And Experiments*, volume 7152 of *LNCS*, pages 146–161, 2012.

[65] Cade Metz. Why apple's switf language will instantly remake computer programming. `https://www.wired.com/2014/07/apple-swift/`, 2004. [Online; accessed August-2019].

[66] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. Technical report, Microsoft Security Response Center, 2019.

[67] Jonathan Moerman. Evaluating the performance of open source static analysis tools. Bachelor's thesis, Radboud University, 2018.

[68] Felipe R. Monteiro, Erickson H. da S. Alves, Isabela S. Silva, Hussama I. Ismail, Lucas C. Cordeiro, and Eddie B. de Lima Filho. ESBMC-GPU a context-bounded model checking tool to verify CUDA programs. *Science of Computer Programming*, 152:63–69, 2018.

[69] Felipe R. Monteiro, Lucas C. Cordeiro, and Eddie B. de Lima Filho. Bounded model checking of C++ programs based on the qt framework. In *Global Conference on Consumer Electronics*, pages 179–180, 2015.

[70] Felipe R. Monteiro, Mário A. P. Garcia, Lucas C. Cordeiro, and Eddie B. de Lima Filho. Bounded model checking of C++ programs based on the qt cross-platform framework. *Software Testing, Verification and Reliability*, 27(3):24, 2017.

[71] Felipe R Monteiro, Mário AP Garcia, Lucas C Cordeiro, and Eddie Batista de Lima Filho. Bounded model checking of c++ programs based on the qt cross-platform framework (journal-first abstract). In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, page 954, 2018.

[72] Steven S. Muchnick. *Advanced Compiler Design And Implementation*. Morgan Kaufmann Publishers Inc., 1st edition, 1997.

[73] Paul Muntean, Jens Grossklags, and Claudia Eckert. Practical integer overflow prevention. *arXiv e-prints*, page arXiv:1710.03720, 2017.

[74] Joseph Neggers and Hae-sik Kim. *Basic Posets*. World Scientific Pub. Co. Inc., 1st edition, 1999.

[75] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2014.

[76] Corina S. Pasareanu and Willem Visser. Verification of java programs using symbolic execution and invariant generation. In *Model Checking of Software*, volume 2989 of *LNCS*, pages 164–181, 2004.

[77] Phillipe Pereira, Higo Albuquerque, Isabela da Silva, Hendrio Marques, Felipe R. Monteiro, Ricardo Ferreira, and Lucas Cordeiro. SMT-based context-bounded model checking for CUDA programs. *Concurrency and Computation: Practice and Experience*, 29(e3934):1–20, 2017.

[78] Prakash Prabhu, Naoto Maeda, and Gogul Balakrishnan. Interprocedural exception analysis for C++. In *European Conference on Object-Oriented Programming*, volume 6813 of *LNCS*, pages 583–608, 2011.

[79] Magnus Ågren. Static code analysis for embedded systems. Master's thesis, University of Gothenburg, Göteborg, Sweden, 2009.

[80] Mikhail Ramalho, Mauro Freitas, Felipe Sousa, Hendrio Marques, Lucas C. Cordeiro, and Bernd Fischer. SMT-based bounded model checking of C++ programs. In *Engineering of Computer Based System*, pages 147–156, 2013.

[81] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. Formal verification of object layout for C++ multiple inheritance. In *Symposium On Principles Of Programming Languages*, pages 67–80, 2011.

[82] Petr Ročkai, Jiří Barnat, and Luboš Brim. Model checking C++ programs with exceptions. *Science of Computer Programming*, 128:68–85, 2016.

[83] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at Google. *Communications of ACM*, 61:58–66, 2018.

[84] Jeremy Siek and Walid Taha. A semantic analysis of C++ templates. In *European Conference On Object-Oriented Programming*, volume 4067 of *LNCS*, pages 304–327, 2006.

[85] Richard L. Sites. Some thoughts on proving clean termination of programs. Technical report, Computer Science Department, Stanford University, 1974.

[86] SoSy-Lab. SV-COMP 2018. `https://sv-comp.sosy-lab.org/2018/`, 2018. [Online; accessed August-2019].

[87] Felipe R. M. Sousa, Lucas C. Cordeiro, and Eddie B. Lima Filho. Verificação de programas c++ baseados no framework multiplataforma qt. In *Encontro Regional de Computação e Sistemas de Informação*, volume 4, pages 181–190, 2015.

[88] Vladimír Štill, Petr Ročkai, and Jiří Barnat. Using off-the-shelf exception support components in C++ verification. In *Software Quality, Reliability & Security*, pages 54–64, 2017.

[89] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 2000.

[90] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[91] S. Thompson and G. Brat. Verification of c++ flight software with the mcp model checker. In *2008 IEEE Aerospace Conference*, pages 1–9, 2008.

[92] Sarah Thompson, Guillaume Brat, and Karl Schimpf. The mcp model checker. `https://ti.arc.nasa.gov/m/pub-archive/1312h/1312%20(Thompson,%20S).pdf`, 2008. [Online; accessed March-2020].

[93] Sarah Thompson, Guillaume P. Brat, and Arnaud Venet. Software model checking of ARINC-653 flight code with MCP. In César A. Muñoz, editor, *Second NASA Formal Methods Symposium - NFM 2010, Washington D.C., USA, April 13-15, 2010. Proceedings*, volume NASA/CP-2010-216215 of *NASA Conference Proceedings*, pages 171–181, 2010.

[94] Heila van der Merwe, Oksana Tkachuk, Brink van der Merwe, and Willem Visser. Generation of library models for verification of android applications. *Software Engineering Notes*, 40(1):1–5, 2015.

[95] Heila van der Merwe, Brink van der Merwe, and Willem Visser. Verifying android applications using java pathfinder. *Software Engineering Notes*, 37(6):1–5, 2012.

[96] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2017.

[97] Andrew Walker, Michael Coffey, Pavel Tisnovsky, and Tomas Cerny. On limitations of modern static analysis tools. In Kuinam J. Kim and Hye-Young Kim, editors, *Information Science and Applications*, pages 577–586, Singapore, 2020. Springer Singapore.

[98] Xiao Xusheng, Balakrishnan Gogul, Ivanŏić Franjo, Maeda Naoto, and Gupta Aarti. NECLA benchmarks: C++ programs with C++ specific bugs. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.410.7773&rep=rep1&type=pdf, 2013. [Online; accessed August-2019].

[99] Jing Yang, Gogul Balakrishnan, Naoto Maeda, Franjo Ivančić, Aarti Gupta, Nishant Sinha, Sriram Sankaranarayanan, and Naveen Sharma. Object model construction for inheritance in C++ and its applications to program analysis. In *Compiler Construction*, volume 7210 of *LNCS*, pages 144–164, 2012.

Appendix A

Experimental Data

Tables A.1, A.2, and A.3 show the experimental data. In those tables, *C* is the number of C++ programs, *LOC* is the number of lines of code off all the benchmarks, *Time* is the total verification CPU time of each test suite, *P* is the number of benchmarks without errors correctly verified, *N* is the number of benchmarks with errors correctly verified, *FN* is the number of benchmarks reported as correct but are incorrect, *FP* is the number of benchmarks reported as incorrect but are correct, *Error* is the number of benchmarks where occurred an internal error during the verification, *TO* is the number of timeouts (i.e., the tool was aborted after 900 seconds), and *MO* is the number of memory-outs (i.e., the tool consumed more than 14GB of memory).

Table A.1: ESBMC $v2.0$ experimental data.

	Test suite	C	LOC	ESBMC $v2.0$							
				Time	P	N	FP	FN	Error	TO	MO
1	templates	23	853	107	7	5	0	0	11	0	0
2	gcc-template	32	1036	387	13	0	0	5	14	0	0
3	cbmc	39	898	5	35	2	0	1	1	0	0
4	algorithm	144	4354	5812	56	59	8	14	5	2	0
5	deque	43	1239	981	20	21	0	2	0	0	0
6	vector	146	6853	1131	74	36	3	14	19	0	0
7	list	72	2292	364	17	21	4	10	20	0	0
8	queue	14	328	290	6	7	0	1	0	0	0
9	priority_queue	15	396	560	8	7	0	0	0	0	0
10	stack	14	286	308	6	6	0	0	2	0	0
11	map	47	1678	698	20	19	0	1	5	0	2
12	multimap	45	1439	428	20	20	0	1	4	0	0
13	set	48	1393	834	18	22	0	7	1	0	0
14	multiset	43	1238	724	14	18	1	6	4	0	0
15	inheritance	51	3460	96	25	18	1	6	1	0	0
16	try_catch	81	4743	43	27	44	3	2	5	0	0
17	stream	66	1831	540	47	12	1	5	1	0	0
18	string	233	4921	7948	106	125	0	2	0	0	0
19	cpp	357	33208	3991	278	39	8	8	22	2	0
		1513	72446	25251	797	481	29	85	115	4	2

Table A.2: DIVINE $v4.0.22$ experimental data.

				DIVINE $v4.0.22$							
	Test suite	C	LOC	Time	P	N	FP	FN	Error	TO	MO
1	templates	23	853	1291	9	8	0	4	2	0	0
2	gcc-template	32	1036	190	26	0	0	4	2	0	0
3	cbmc	39	898	215	35	0	0	0	4	0	0
4	algorithm	144	4354	21867	1	70	0	73	0	0	0
5	deque	43	1239	7600	0	21	0	22	0	0	0
6	vector	146	6853	20681	1	34	0	91	20	0	0
7	list	72	2292	9372	0	34	0	36	2	0	0
8	queue	14	328	2466	0	7	0	7	0	0	0
9	priority_queue	15	396	1953	1	7	0	7	0	0	0
10	stack	14	286	1964	0	7	0	6	1	0	0
11	map	47	1678	6844	0	22	0	25	0	0	0
12	multimap	45	1439	2890	0	22	0	23	0	0	0
13	set	48	1393	5748	1	22	0	24	1	0	0
14	multiset	43	1238	6012	0	21	0	22	0	0	0
15	inheritance	51	3460	3914	3	13	0	24	11	0	0
16	try_catch	81	4743	4704	5	37	4	25	10	0	0
17	stream	66	1831	8269	1	13	0	51	1	0	0
18	string	233	4921	27683	34	125	0	74	0	0	0
19	cpp	357	33208	43292	18	27	1	244	67	0	0
		1513	72446	176957	135	490	5	762	121	0	0

Table A.3: LLBMC $v2013.1$ experimental data.

				LLBMC $v2013.1$							
	Test suite	C	LOC	Time	P	N	FP	FN	Error	TO	MO
1	templates	23	853	1825	8	8	1	5	0	1	0
2	gcc-template	32	1036	2	26	0	0	5	1	0	0
3	cbmc	39	898	<1	34	0	0	1	4	0	0
4	algorithm	144	4354	12549	62	61	1	3	6	10	1
5	deque	43	1239	7086	16	17	0	0	2	6	2
6	vector	146	6853	6611	87	38	1	3	9	4	4
7	list	72	2292	1333	8	29	5	29	0	0	1
8	queue	14	328	32	6	7	0	1	0	0	0
9	priority_queue	15	396	9008	2	3	0	7	0	3	0
10	stack	14	286	32	6	7	0	0	1	0	0
11	map	47	1678	322	8	17	5	17	0	0	0
12	multimap	45	1439	951	3	19	3	19	0	0	1
13	set	48	1393	315	5	20	2	20	1	0	0
14	multiset	43	1238	599	3	20	1	19	0	0	0
15	inheritance	51	3460	98	23	12	0	3	13	0	0
16	try_catch	81	4743	–	–	–	–	–	–	–	–
17	stream	66	1831	<1	17	13	0	35	0	1	0
18	string	233	4921	1	6	121	4	102	0	0	0
19	cpp	357	33208	3675	213	21	8	47	65	1	2
		1513	72446	44438	533	413	31	316	183	26	11