

FACULTY OF INFORMATICS, MASARYK UNIVERSITY



Analysis of Parallel C++ Programs

PHD THESIS

Vladimír Štill

Supervisor:

prof. RNDr. Jiří Barnat, Ph.D.

Brno, 2020

Declaration

Thereby I declare that this thesis is my original work, which I have created on my own. All sources and literature used in writing the thesis, as well as any quoted material, are properly cited, including full reference to its source.

Vladimír Štill

Supervisor: prof. RNDr. Jiří Barnat, Ph.D.

Abstract

Parallel software offers a promise of full utilisation of modern hardware. Unfortunately, building a parallel program presents some additional challenges for the programmers. In this thesis, we introduce some improvements to the analysis of parallel C++ programs. In particular, we aim to help with the discovery of hard-to-find bugs.

As our first contribution, we deal with some of the problems related to analysis of high-level programming languages, including their advanced features and standard libraries. We consider this topic important as comprehensive language support makes the analysis tool more usable by programmers in practice. Comprehensive language support is not an easy task. However, we show it is still manageable with the right combination of reuse of existing execution-oriented components and design of new, verification-oriented ones. In this work, we deal with C++ support for the DIVINE verifier in general, and its support for C++ exceptions in particular.

Our second contribution is a novel approach to the analysis of programs running under the relaxed memory model of Intel and AMD x86 processors. These processors can delay memory stores after independent loads which can lead to peculiar behaviour of parallel programs. This behaviour is often hard to keep track by programmers and therefore can be a source of subtle bugs. We propose a novel way in which the verifier can simulate relaxed memory. Our method aims to minimise introduced nondeterminism and therefore increase the overall performance of the verification.

As our last contribution, we introduce a method for checking local nontermination of parallel programs, i.e., for detection of sections of the program that are supposed to terminate but do not. The local nontermination can be used to detect problems in programs that do not terminate but have parts that must do so (e.g., a server might have a function for handling a request, and this function must always terminate). Our method uses lightweight annotations to mark parts of the program which must terminate. It is not limited to the use of particular synchronisation primitives, but it is only complete for programs which have finite state space (such programs can have infinite behaviour in which specific state is repeated infinitely). Even under this limitation, we believe this technique can help with the design of high-performance parallel algorithms and data structures.




The contributions presented in this thesis are implemented in an open-source software model checker DIVINE and are accompanied by experimental evaluation.


Keywords

program analysis, software verification, C++, DIVINE, LLVM, parallelism, concurrency, relaxed memory, memory models, exceptions, termination, program transformation, implementation

Acknowledgments

I want to thank the members of the ParaDiSe laboratory for nice, even if not always particularly work-friendly environment. In particular, I want to thank my supervisor, Jiří, for discussions about my research topics, help with papers, and trying to keep me on the track while also grudgingly respecting that I cannot devote all my time to research. I want to thank Mornfall for his consultations, programming insights, inputs to research topics, and help with papers, especially in the first half of my PhD and in my bachelor and master studies. I want to thank Nikola for being an inspiration in both depth and breadth of knowledge, for the energy he puts into teaching, and for the long discussions on various topics. I want to thank Martin J. for being an inspiration in what a PhD student can achieve, for his indispensable insights to teaching materials and also for the long discussions. Furthermore, I want to thank Red Hat for the support of my research. I am also thankful for the opportunity to teach during my studies at the faculty – it was an important contributor in my career choices and allowed me to meet many interesting people.

However, this thesis would not exist without many people who did not participate in the research and without additional opportunities I had. I want to thank my closest family for the support they gave me during the long course of my studies. My thanks go to many members and former members of ParaDiSe and Formela laboratories, many of whom are much more than just colleagues. My thanks also go to my friends in the Friends of Nordic Animals Association³⁹ and to this association in general. Furthermore, I am thankful to Instruktoři Brno for providing the much-needed counter-balance to my research activities in the last hard year. In particular I want to thank , , and  for organizing the 2019/2020 course, to all the participants of this course and to all the people who organized its events. Big thanks go to Martin U. and Katka S. for really listening to me when I needed it a lot. Finally, I am pretty sure this list is not complete... well, a complete list would probably be too long anyway.

 Spolek přátel severské
zvěře



Contents

1	Introduction	1
1.1	Problem Statements and Contributions	2
1.2	Supplementary Materials	6
1.3	Thesis Structure	6
2	Preliminaries	7
2.1	Program Analysis	7
2.2	Parallelism & Threading Model	11
2.3	Programming Languages Used in this Thesis	20
2.4	DIVINE	26
3	State of the Art	31
3.1	Explicit-State Model Checking	31
3.2	Stateless Model Checking	35
3.3	Symbolic and Concolic Execution	39
3.4	Bounded Model Checking & Other Symbolic Techniques	42
3.5	Other Approaches to Program Analysis	47
4	Improvements in Analysis of Realistic Programs	51
4.1	Component Reuse in Program Analysis	52
4.2	Component Reuse in DIVINE	55
4.3	C++ Exceptions in DIVINE	57
4.4	Exceptions in C++	59
4.5	Execution of LLVM programs	61
4.6	The LLVM Transformation	63
4.7	The Unwinder	65
4.8	Related Work	68
4.9	Evaluation	69
4.10	Conclusion	72
5	Analysis of Programs Under the x86-TSO Relaxed Memory Model	75
5.1	Motivation and Introduction	75
5.2	Conventional Semantics of x86-TSO	76
5.3	x86-TSO in DIVINE	77
5.4	Evaluation	83
5.5	Related work	85
5.6	Conclusion	87

6	Local Nontermination analysis for Parallel Programs	89
6.1	Motivation and Introduction	89
6.2	Resource Sections	91
6.3	Local Nontermination	92
6.4	Detection of Nontermination	95
6.5	Evaluation	101
6.6	Related Work	104
6.7	Conclusion	106
7	Conclusion	109
7.1	Contributions	109
7.2	Future Work	111
A	Published Papers	113
A.1	Most Significant Papers	113
A.2	Other Papers	115
B	Bibliography	117

Chapter 1

Introduction

Our modern world depends on software in many aspects. In the morning, software on our mobile phone can wake us up. Then we often travel to work either by car or by public transport. In either case, there is probably software in the vehicle, including life-critical software in engine control and safety systems such as ABS (brakes anting-blocking system). In work, many people use computers to do their job and to communicate with coworkers and the rest of the world. In the meanwhile, our mobile phone and the Internet accompanies us on almost every step. We also use much infrastructure which can have important software components in its control systems – including energy grids, smart traffic signs, and flight control.

Any of this plethora of software system can contain bugs. These bugs can range from minor ones that cause inconvenience only, to safety-critical problems that can cause deaths of many people. For this reason, ensuring software correctness is a desirable goal pursued by software developers and testers, as well as researchers. Many techniques provide assistance to both developers and testers, with a wide range of capabilities and ease of use. At the basic level, developers can leverage type system, especially if they use strongly statically typed programming language, and they can use compiler warnings and linters¹ to check for mistakes and potential problems in the code. Code is also routinely tested with a large variety of test types, starting from unit tests that test small parts of the program in isolation and continuing to tests of functionality of the whole system.

These testing and code analysis techniques are wide-spread, can discover significant amounts of bugs, and are often reasonably easy to use for developers. However, these techniques cannot prove the absence of bugs, and certain types of bugs are notoriously hard to discover by these testing techniques. For example, testing parallel software is inherently hard due to thread interactions which can cause different executions of the same program (or function) to produce different results. The program can run right in almost all cases, but can occasionally fail, maybe once in a few minutes, maybe once in a few months. A conventional approach to this problem is using stress testing. For example, a thread-safe data structure might be stress tested by performing millions of operations with it from many threads. Such stress tests are designed to increase the likelihood of triggering a bug in

¹ Lightweight analysis tools that try to find likely bugs. They rely on approximation and usually perform more thorough analysis than compilers do for warnings, but still prefer speed and a low number of spurious error reports to the discovery of hard-to-find bugs.

the test, but they still cannot show its absence and can miss truly rare bugs. Even worse, tests of parallel software are often nondeterministic on buggy software, i.e., the same test of the same software version can sometimes succeed and sometimes fail due to different interleaving of threads in different executions. This behaviour might make bug fixing extremely difficult. For example, a test may fail in a nightly automatic build, and therefore the developers know there is a bug in the program, but they are unable to reproduce the bug, and the test result might not be sufficient to find what the bug is without reproducing it. Similarly, it is hard to ensure by testing that a (parallel) program terminates in all cases, or that it works correctly even on hardware with relaxed memory (which can, for example, delay stores to the main memory).²

² Consider this code:

```
int x = 0;
int y = 0;

void t0() {
    y = 1;
    int a = x;
}

void t1() {
    x = 1;
    int b = y;
}
```

Where `t0` and `t1` are executed by different threads.

Intuitively we could expect that $a = 0 \wedge b = 0$ is not a possible outcome.

However, under the common x86 architecture, this outcome can happen due to delayed stores.

To make the discovery of hard-to-find bugs easier, many techniques were proposed. These include both testing-based methods, such as race detectors and thread sanitisers or record and replay debuggers, and methods based on *formal verification* including theorem proving, symbolic execution and various variants of model checking.³ In this work, we will focus on methods with a more formal basis that are capable of the discovery of hard-to-find bugs. However, for these methods to be useful to programmers, they must not only be theoretically capable of bug discovery, but they must also be reasonably convenient to use. For example, it is highly desirable that the program analysis tool can analyse the program directly, without the need to create a model of the program that is to be analysed (many, especially older, techniques for analysis of parallel programs require models). For this reason, we will focus on analysis tools which work with *realistic programs*, i.e., programs written in high-level programming languages which can be both executed on the desired platform and analysed without a separate modelling step.

³ The distinction between testing and formal verification is not clear and many techniques derived from formal verification are unable to prove correctness. This blurring is further increased by methods like stateless model checking that is also presented under the name systematic concurrency testing.

[Bar+17] Baranová et al., “Model Checking of C and C++ with DIVINE 4”.

[Roč20] Ročkai, “DIVINE 4”.

1.1 Problem Statements and Contributions

In this work, we will describe several techniques that improve analysis of realistic parallel programs in the DIVINE model checker [Bar+17; Roč20], and we will compare them to many other methods that aim at the discovery of hard-to-find bugs in parallel programs. In our analysis, we will focus primarily on programs written in C and C++. Nevertheless, analysis of other real-world high-level imperative programming languages (such as Java, C#, Rust and many other) should follow similar principles.

The work presented in this thesis is also accompanied by an open-source implementation in DIVINE. At its core, DIVINE is an explicit-state model checker extended with data abstraction capabilities. It can be used to discover large classes of bugs, including assertion violations, use of uninitialised memory, bad memory accesses, memory leaks, and concurrency related bugs such as deadlocks.

1.1.1 Analysis of Realistic Programs

Analysis of realistic (parallel) programs is a hard task because these programs have many features that are complex to handle by an analysis

tool. First, the syntax of the programming language is often complex – modern programming languages such as C++, C#, Java or Python are designed to be expressive and well usable by programmers and are usually not easy to parse and process. Furthermore, the language can have many features which are hard to analyse even when the code is understood. For example, dynamic memory complicates program analysis because the amount of used memory and identifiers of memory locations are not known beforehand. Pointer arithmetic and memory unsafety of programming languages such as C and C++ further complicate the handling of memory. Even procedure calls and recursion, which are available in virtually any general-purpose programming language, make some kinds of program analysis harder.

Realistic programming languages are also defined not only by the language syntax and semantics but also by their libraries. The programming language usually comes with a standard library that provides basic abstractions and operations that most of the programmers will expect to work. Program analysis tools must also understand these features if the programmers are to use them. Even relatively minimalistic standard libraries, such as the C standard library, contain features like memory allocation and deallocation, basic string manipulations, math operations, basic input and output (including basic filesystem access), and since C11 also threads and their synchronisation. Most standard libraries are even larger and more complex – they contain various data structures, algorithms for working with them, and often abstractions over filesystem and network access.

For a program analysis tool to be useful to programmers, it should be able to process the code they are creating with minimal extra effort. This means it must be able to handle complex language features and libraries. For example, tools which understand basics of the C language, but do not allow dynamic memory allocation, are not very useful for programmers, as they will probably not be able to use them to analyse their usual programs. On the other hand, if the language support is good enough, it enables the programmer to use the analysis tool directly on their program or its fragments, both during the development and for older code.

Apart from the ease of use, program analysis without modelling or simplification also increases the reliability of the analysis results – if the analysed program differs from the program which is actually executed, the difference can hide problems or introduce new, spurious bugs.

Our Contribution In Chapter 4 we present a wider look at the problem of analysis of realistic programs from the point of features of programming languages and their libraries. We show that library reuse can be a viable approach for program analysers with good support for language features and that it can significantly simplify support for languages with complex libraries such as C++. We also identify functionality which is a good candidate for verification-specific modelling instead of reuse. Finally, we focus on the particular case of exception support for C++ in DIVINE and how it can be achieved

[Ram+13] Ramalho et al., “SMT-Based Bounded Model Checking of C++ Programs”.

[GBHR20] Garzella et al., “Leveraging Compiler Intermediate Representation for Multi- and Cross-Language Verification”.

[Vis+03] Visser et al., “Model checking programs”.

[KT14] Kroening et al., “CBMC – C Bounded Model Checker”.

[CKS19] Cordeiro et al., “JBMC: Bounded Model Checking for Java Bytecode”.

[ŠRB17] Štill et al., “Using Off-the-Shelf Exception Support Components in C++ Verification”.

by a combination of library reuse and creation of verification-specific libraries.

Other works focusing on language support in program analysis tools include, for example, [Ram+13; GBHR20]. Exceptions are also supported in many Java tools and some tools with C++ support [Vis+03; KT14; CKS19]. However, to the best of our knowledge, our support of C++ exceptions is the only complete support for C++ exceptions in a comparable tool. Furthermore, our method of implementation of exceptions is rather generic and leverages existing language-specific exception-handling code from C++ standard library implementation. Therefore, it should be possible to adapt it to other programming languages with exception support with reasonable effort.

This contribution was first presented in [ŠRB17]. Chapter 4 is further extended with unpublished content about language support in general.

1.1.2 Parallelism and Relaxed Memory

To fully utilise the capabilities of modern hardware, programmers are encouraged to write parallel software. However, parallelism adds more complexity both for the author of the code and for the analysis tool as it has to be able to handle representation of threads and synchronisation in the given programming language and the semantics of parallel execution of the program. Furthermore, relaxed memory can come into play with parallelism. Modern processors use cache memories and out-of-order execution to improve their speed and hide speed difference between the processor’s cores and the main memory. For efficiency reasons, these mechanisms are often not transparent to the programmer and can be observed by multi-threaded programs, yielding possible executions that violate ordering of actions in their threads (for example, a write to a memory can be delayed past an independent read). When this behaviour is observable on a hardware platform, we say that the given hardware platform has relaxed memory.

Relaxed memory adds substantial complexity to the program analysis – an analysis tool has to be able to understand the particular relaxed memory model (different platforms exhibit different behaviour) and the amount of resources required to run the verification is also often significantly increased. We believe that analysis under relaxed memory models is an important topic, in particular in the context of lock-free programs.

Our Contribution Chapter 5 shows how efficient support for the memory model of the common Intel and AMD x86-64 processors was added to the DIVINE verifier. In our work, we leverage the LLVM infrastructure to implement support for relaxed memory as a preprocessing step, without the need to modify the verifier itself (provided it has support for nondeterministic choice). Furthermore, we show that the amount of nondeterminism in the resulting program can be limited by careful design of data structures used to simulate the behaviour of x86-TSO store buffers. The decreased nondeterminism directly trans-

lates to smaller state space and therefore, faster and less memory-hungry program analysis.

Analysis of programs under various relaxed memory models is an active research area with a lot of related techniques, including [AKNT13; ZKW15; Abd+17; KLSV17; AAJN18]. Our work, which was originally presented in [ŠB18], presents a significant improvement over existing explicit-state techniques and complements well techniques based on bounded or stateless model checking.

1.1.3 Local Nontermination of Parallel Programs

An important part of the correctness of programs is that they eventually do the work they are supposed to do. This implies the program should terminate or, often in the case of parallel or event-driven programs, that it should handle each request within a finite amount of time. For example, a server is often running an infinite loop that spawns request handlers, and each of these request handlers must terminate. Similarly, a critical section of a program is usually required to terminate to ensure the program does not get stuck. Lock-free programs can sometimes rollback actions and it is important to check that the action will eventually finish.

Therefore, termination checking (and more generally checking of liveness properties) is an important companion to safety checking⁴ in the pursuit of correct parallel programs. Furthermore, it is often not sufficient to check that the whole program terminates. It is more desirable to check that some designated part of the program terminates (e.g., critical section, event handler).

Our Contribution In Chapter 6, we introduce a generic method for detection of nontermination caused by communication between threads. It uses lightweight annotations in the code to mark parts of the program that must terminate, together with a set of pre-defined parts of programs that must terminate which allow it to be used with common synchronisation primitives out of the box. Our method can also be applied to deliberately nonterminating programs (i.e., daemons, services) in which it can detect parts that should terminate but do not terminate.

Termination of parallel programs is less explored than for the sequential case. Nevertheless, there are many existing works, including specialized tools like [CC14; Aga+10; BH05] (which are specialized to given synchronisation primitives) and [Cha+05; DIS99] (which can detect global deadlocks). More general tools include those based on modular proofs [CPR07; PR12] or loop detection [ABEL12]. Our work, originally presented in [ŠB19], is based on state-space exploration and focuses primarily on nontermination caused by thread communication – it can handle arbitrary synchronization primitives but does not handle symbolic data so far. Furthermore, one of its main novelties is that it can detect nonterminating parts of programs with user-defined granularity (e.g., a specified function or its part).

[AKNT13] Alglave et al., “Software Verification for Weak Memory via Program Transformation”.

[ZKW15] Zhang et al., “Dynamic Partial Order Reduction for Relaxed Memory Models”.

[Abd+17] Abdulla et al., “Stateless model checking for TSO and PSO”.

[KLSV17] Kokologiannakis et al., “Effective Stateless Model Checking for C/C++ Concurrency”.

[AAJN18] Abdulla et al., “Optimal Stateless Model Checking under the Release-Acquire Semantics”.

[ŠB18] Štill et al., “Model Checking of C++ Programs Under the x86-TSO Memory Model”.

⁴ Safety checking aims to show that the program does not perform any forbidden action. Examples include assertion safety (absence of violation of *assertions* – programmer specified properties that should hold at the given point in the code), memory safety (which includes correct access to arrays, correct handling of dynamic memory, and absence of stack overflows), or absence of use of undefined values (in programming languages such as C and C++ that can work with uninitialised memory).

[CC14] Cai et al., “Magi-clock: Scalable Detection of Potential Deadlocks in Large-Scale Multithreaded Programs”.

[Aga+10] Agarwal et al., “Detection of Deadlock Potentials in Multithreaded Programs”.

[BH05] Bensalem et al.,
“Scalable dynamic deadlock analysis of multi-threaded programs”.

[Cha+05] Chaki et al.,
“Concurrent software verification with states, events, and deadlocks”.

[DIS99] Demartini et al., “A deadlock detection tool for concurrent Java programs”.

[CPR07] Cook et al., “Proving Thread Termination”.

[PR12] Popeea et al.,
“Compositional Termination Proofs for Multi-threaded Programs”.

[ABEL12] Atig et al.,
“Detecting Fair Non-termination in Multi-threaded Programs”.

[ŠB19] Štill et al., “Local Nontermination Detection for Parallel C++ Programs”.

1.2 Supplementary Materials

Supplementary materials for this work can be found on vstill.eu/phd. This web page contains links to the relevant publications this work is based on and to their respective supplementary pages.

1.3 Thesis Structure

After this introduction, Chapter 2 gives the preliminaries need for the rest of the work, including an introduction to program analysis, parallelism and relaxed memory models, C++, and DIVINE. Chapter 3 presents state of the art in the program analysis of realistic parallel programs. The next three chapters present the main contributions of this work: analysis of realistic programs and C++ exceptions in Chapter 4, support for the x86-TSO memory model in Chapter 5, and nontermination detection in Chapter 6. Chapter 7 then concludes the main body of this work.

The list of my published results can be found in Appendix A.

Chapter 2

Preliminaries

This chapter gives an introduction to program analysis, parallel programs, and relaxed memory models. We will also introduce the C++ programming language as it is the language we are primarily targeting and the DIVINE model checker as all the contributions are implemented in DIVINE.

2.1 Program Analysis

We will not be concerned with simple program analysis techniques like type-checkers and linters. Instead, we will focus on techniques based on formal verification, that is techniques which can provide some formally described guarantees about the program if they succeed. These techniques usually require a program and some specification the program should adhere to and can check if that is the case. We will also focus mostly on techniques which can be directly applied to programs written in some mainstream programming language (i.e., techniques which do not require special verification-oriented languages as their inputs).

Program analysis techniques can be broadly divided into two different areas, *automatic techniques* that, provided a program and its specification, should produce a result automatically without further assistance, and *human-assisted techniques* which require a substantial human effort during the analysis. Examples of the first area are symbolic execution and various model checking techniques, while theorem provers are an example of the latter kind. As we focus on techniques that should be usable by programmers without a substantial background in formal logic, we will focus on the automatic tools.

2.1.1 State Space

To be able to define program properties that should be checked and to describe various analysis methods, we first need to define the *state space* of a program.

Definition 2.1: State Space

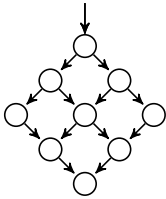
The *state space* of a program is a directed multigraph with (optionally) labelled edges that describes all the ways in which the program can be executed. The vertices of the state space multigraph are called

states (of the program). Each state represents a certain point in the execution of the program – it can be described by a snapshot of the program (its memory, program counters and stacks of all its threads, ...). States v_1, v_2 are connected by an edge in the state space if v_2 can be reached from v_1 by a single step of the program. The edge can be optionally labelled, for example by the statements executed in each step, or by selected actions specified by the program or verification tool (such as *error* label to indicate an error occurs on this edge or an *accepting* label that can be used in the automata-based approach to LTL model checking). The state space contains all the states of the program reachable from an initial state (an initial configuration of the program).

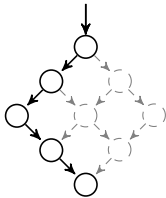
The input for the analysis is usually not the state space graph. Typically, the state space is specified by its implicit representation – the program code, or possibly a function that describes the initial states and how to get from one state to its successors.

In practice, the state space of a program can be very large or infinite, and it is useful to be able to consider only a representative part of the state space.

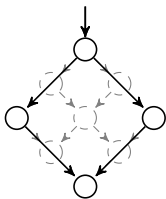
¹ For example, consider the following state space:



A corresponding reduced state space might be:



Another possibility is:



[Pel93] Peled, “All from one, one for all: on model checking using representatives”.

[FG05] Flanagan et al., “Dynamic Partial-order Reduction for Model Checking Software”.

[RBB13] Ročkai et al., “Improved State Space Reductions for LTL Model Checking of C & C++ Programs”.

Definition 2.2: Reduced State Space

A *reduced state space* of a program is a subset of the state space in which states are subset of states of the original state space and edges connect states between which there is a directed path in the original state space such that all the internal states of this path are not contained in the reduced state space.¹

Several techniques can be used to construct reduced state space in such a way that a given property (or a class of properties) holds in the reduced state space if and only if it holds in the original state space. For example, partial order reduction [Pel93], dynamic partial order reduction [FG05] or τ + reduction [RBB13].

On top of the reductions, it is also possible to build an *abstraction* of a state space, i.e., a state space that preserves some of the properties of the original but abstracts away some of its complexity. These abstractions are usually built from the implicit representation of the program. Abstractions are often used to reduce large or infinite state spaces to sizes that can be managed. Usually, abstractions are designed such that results derived from the abstract state space can be used to obtain some information about the original state space.

Definition 2.3: Over- and Under-Approximations in Abstraction

Suppose we have an abstracted state space \hat{S} that corresponds to a (concrete) state space S . Suppose further that we are interested in the reachability of states that satisfy some property P that can be evaluated both in S and \hat{S} .

- We say that \hat{S} is an *over-approximation* of S if the fact that no state that satisfies P is reachable in \hat{S} implies that no such state is reachable in S (i.e., \hat{S} can contain more behaviour that leads to such a state).

- We say that \hat{S} is an *under-approximation* of S if the fact that a state that satisfies P is reachable in \hat{S} implies that there is a state that satisfies P reachable in S (i.e., \hat{S} can contain less behaviour that leads to such a state).

Abstractions are also connected with the notion of *refinement*. Often a program is over-approximated, then analysed, and if an error is found, this error is validated. If the error is valid, then there is a real error in the original program. However, if the error is not valid, the error trace is used to *refine* the abstraction, i.e., make it more precise to rule out this spurious error (and preferably also similar spurious errors). This technique is often called *Counterexample-Guided Abstraction Refinement* (*CEGAR*) [Cla+00; Kur95].

An example of an abstraction technique is *predicate abstraction* [CU98]. Instead of tracking original values of variables, program abstracted with predicate abstraction tracks only validity of predicates over the variable values. For example, it might track predicates such as $(x > 0)$ or $(y \leq x)$. Refinement with is usually done by addition of new predicates.

Finally, we will need the following state-space-related definitions.

Definition 2.4: Finite State Space

We say that a program has a *finite state space* if the number of vertices and edges in the state space multigraph is finite. Otherwise, we say the program has an *infinite state space*.

Definition 2.5: Run

A *run* in the state space is a (possibly infinite) path in the state space (or reduced state space) that starts in the initial state. That is, a run is a sequence of states $\sigma = s_0, s_1, \dots$ such that s_0 is the initial state and for each consecutive pair of states s_i, s_{i+1} there is an edge from s_i to s_{i+1} in the corresponding (reduced) state space. States can be repeated in a run and a run can be infinite.

2.1.2 Program Properties

To be able to find errors in programs, an analysis tool has to have some specification of program correctness – a *property* that should be checked. In this section, we will introduce several categories of program properties together with examples of particular properties.

Safety Properties are properties that can be checked locally – for each state of the program, we can determine if the state satisfies or violates the given property. Therefore, to conclude the program is error-free under given safety property, it suffices to show that no state violates the property.

Examples of safety property include *memory safety* (every time we access a particular part of memory, this memory is allocated and accessible to the program), *assertion safety* (a program does not violate any of the assertions in the source code), and *control-flow definedness* (every time a conditional jump is performed, the values the jump is based on must have well-defined value).

[Cla+00] Clarke et al., “Counterexample-Guided Abstraction Refinement”.

[Kur95] Kurshan, “Computer-aided Verification of Coordinating Processes: The Automata-theoretic Approach”.

[CU98] Colón et al., “Generating finite-state abstractions of reactive systems using decision procedures”.

Some program properties can be checked locally only if sufficient information is kept in the program state. For example, checking *the absence of memory leaks* requires that it is possible to enumerate all allocated objects and all objects to which there are usable pointers. Similarly, mutex-related deadlocks can be detected by safety analysis if the program keeps track of the graph of waiting for mutexes.

Temporal properties Not all properties can be directly described as safety properties. For example, we can have a property stating “every time a button is pressed, the elevator will eventually drive to the floor the button was pressed on”. Such a property cannot be checked directly in one state of the program, but it is a property of *runs* of the program.

Various temporal logics can be used to describe such properties, for example LTL, CTL, CTL* [BK08] or μ -calculus [Koz82].

[BK08] Baier et al., “Principles of Model Checking”.

[Koz82] Kozen, “Results on the propositional μ -calculus”.

Termination Properties A distinguished kind of temporal properties are termination properties, which are properties which allow us to specify that a program must (or must not) terminate, or that some of its parts must (or must not) terminate.

With tools that check for termination, we often distinguish *termination analysis*, which is an analysis that aims to prove termination but is often unable to conclude that a program does not terminate, and *nontermination analysis*, which aims to prove that the program does not terminate. This distinction is useful as the heuristics used for proving termination and nontermination are often different, and therefore a single analyser might not possess both capabilities.

2.1.3 Hardware and Platform Related Considerations

When a program is executed on given hardware, its behaviour might depend on the concrete features of the hardware. Therefore, for the program analysis to be usable, one must know what hardware model(s) the program analysis tool adheres to and what is their relation to the real hardware the program will be executed on. In practice, it is often hard to prove that a program will be correct on *any* hardware on which it will be possible to compile it.

For example, the size (and therefore precision) of the standard C/C++ type `int` might depend on the hardware the program is using: on an 8-bit or 16-bit embedded microprocessor, `int` will likely be a 16-bit type with a maximum value of $2^{15} - 1$, while on a common 64-bit (or 32-bit) computer it will be a 32-bit number with the maximum value of $2^{31} - 1$. Therefore, a code that causes a bug due to integral overflow on a 16-bit embedded microprocessor might be correct on a 64-bit machine.

Another example concerns the *alignment* of values in memory. Often it is possible to address single bytes in memory, but certain data types (larger than a byte) are only allowed to start on addresses divisible by a given *alignment*. For example, an alignment of a 32-bit `int` type might be 4 bytes (32 bits). Depending on the hardware platform, reading and writing unaligned values might work (e.g., on x86 and x86-64), might trigger an error (e.g., on some ARM processors) or the address might be

silently rounded to the nearest aligned address (e.g., on some embedded ARM processors). Therefore, the behaviour of the same program with an unaligned read can differ drastically depending on the platform on which it is executed.

Furthermore, some features of the program are affected not only by the hardware, but also by the operating system (or more generally the platform) for which it is compiled. For example, the calling conventions of C programs compiled on Windows and Linux differ even on the same hardware. Similarly, sizes of data types might depend on the platform – on Windows C++ `long` is 32 bits long even on 64-bit systems (and therefore cannot be used to hold values of pointers), while it is 64 bits long on 64-bit Linux systems.

2.2 Parallelism & Threading Model

Parallelism is an essential part of programming high-performance software that can fully take advantage of the current hardware. However, parallelism comes with additional problems not present in the development of sequential software. Namely, due to the need for synchronisation, it is significantly harder to create correct parallel software compared to the development of sequential software. Furthermore, it is also hard to create parallel software that scales well with the number of processors (or processor cores) available. Improving scalability usually makes the problem of correctness even harder – it often requires more fine-grained (and therefore error-prone) synchronisation. For example, fine-grained locking or lock-free programming can lead to high-performance code, but checking that the code uses synchronisation properly is significantly harder than for code that uses a few locks with well defined regions of shared memory they guard.

Furthermore, relaxed memory behaviour comes into play once shared variables are not only accessed in critical sections (i.e., protected by mutexes). As processor manufacturers strive to increase the speed of processors, they have introduced various optimisations into the memory infrastructure to avoid waiting for the relatively slow main memory. These optimisations can be (and in most processors are) visible to a parallel program and result in relaxed memory behaviour. For example, on x86 processors (which are used in most laptop, desktop and server computers) a memory store can be delayed and appear, from the point of view of the other threads, later than a subsequent load. Some processors (for example POWER and higher-performance ARM processors) allow even more reordering, for example, reordering of independent loads.

In this section, we describe the basic model of parallelism we assume for our programs and follow some of its implications. We also outline the basics of relaxed memory behaviour and memory models which describe it.

2.2.1 Basic Threading Model

We assume that a program consists of one or more threads that interact using shared memory.² We also assume that execution of a thread can be interrupted at any point by a *thread scheduler* (of the operating system

² Other communication methods can be emulated using shared memory. Furthermore, many programming languages, including C++, have no native support for other communication schemes.

on which the program runs). The execution will be later eventually resumed (unless the program exits in the meantime or the thread is *blocked* infinitely). A thread can be *blocked* if it waits for a resource provided by the operating system, for example, availability of a lock or input from an input device. If a thread cannot be resumed because it is blocked and does never cease to be blocked we say there is a *deadlock*.

Definition 2.6: Deadlock, Partial Deadlock, Global Deadlock

A *deadlock* happens if a scheduler never allows a thread to run because it waits for a resource that never becomes available.³

Sometimes, when we need to distinguish between all the threads being blocked and only some of them being blocked, we will use the notion of *partial deadlock* to signify a situation in a program where some threads are blocked (in deadlock), but others can still proceed and *global deadlock* for a deadlock which blocks all threads. Usually, we will just use *deadlock* to refer to *partial deadlock*.

It is important to note that our definition of deadlock does not include *busy waiting* – i.e., a situation in which a thread is executing a loop that tests that some event occurred. Busy waiting can be used for example to implement exclusive sections without operating system support.

Definition 2.7: Livelock

Livelock is a situation in which a thread is allowed to run (by the scheduler), but does not proceed in any meaningful way because it executes a loop that is waiting for an event that never happens.⁴

Interleaving of Threads In practice program threads can run concurrently, i.e., multiple cores of the processor can execute different threads at the same instant in time. Furthermore, the scheduler can map threads to processor cores arbitrarily, and it can even change the core that executes a given thread. It is not practical (and not necessary) to simulate this behaviour when performing an analysis of a parallel program. Instead, we consider that threads are interleaved (*interleaving semantics*) or we consider execution based on some relaxed memory model.

Definition 2.8: Interleaving Semantics of Threads

With the *interleaving semantics*, we assume that all possible executions of a parallel program can be obtained by interleaving. That is, at any point in the execution of the program, we consider which threads are allowed to run and explore all possible selections. After a thread performs an action, we again consider all possible actions of all threads and so on.

2.2.2 Relaxed Memory Models

A *memory model* describes the behaviour of memory operations on a given platform. A *relaxed memory model* describes such behaviour on a platform with relaxed memory. When considering relaxed memory models, it is also useful to consider a memory model that is not relaxed

³ For example, suppose a program with threads `t0` and `t1`:

```
mutex mtx0, mtx1;
```

```
void t0() {
    unique_lock(mtx0);
    unique_lock(mtx1);
    /* ... */
}
```

```
void t1() {
    unique_lock(mtx1);
    unique_lock(mtx0);
    /* ... */
}
```

If `t0` locks `mtx0` and `t1` locks `mtx1` then each of the threads will attempt to lock the mutex already owned by the other thread. Therefore, the threads are in deadlock – neither of them can proceed and both are blocked.

⁴ Suppose a program with threads `t0` and `t1`:

```
atomic<bool> flag;
```

```
void t0() {
    while (!flag) {
        /* wait */
    }
    /* ... */
}
```

```
void t1() {
    flag = true;
    /* ... */
    flag = false;
    /* ... */
}
```

If `t0` starts executing the `while` loop only after `t1` sets `flag` to `false` the loop will never end, and `t0` is in livelock – it can execute, but does not proceed (provided `flag` is not set again).

and corresponds to the interleaving semantics of threads. This memory model is *sequential consistency*.

Definition 2.9: Sequential Consistency

Sequential consistency is a memory model corresponding to the interleaving semantics of threads. Under sequential consistency, all effects of memory manipulating instructions are visible to all threads immediately, and no instruction reordering is observable.

The operations semantics of sequential consistency is given by a machine that has no instruction reordering, and all memory operations are atomic and access memory directly, without any caches. Sequential consistency is the strongest memory model (it is not relaxed at all). Therefore, any behaviour observable under sequential consistency will also be possible under a relaxed memory model, but a relaxed memory model can exhibit additional behaviour.

In practice, sequential consistency is not sufficient with modern processors that exhibit relaxed memory behaviour. The relaxed behaviour of processors arises from optimisations in cache consistency protocols and observable effects of instruction reordering and speculation. The effect of this behaviour is that memory-manipulating instructions can appear to be executed in a different order than the order in which they appear in a thread's code, and their effect can in some cases even appear to be in a different order on different threads. For efficiency reasons, most modern processors (except for simple ones in embedded microcontroller and low-cost mobile devices) exhibit relaxed behaviour. The extent of this relaxation depends on the processor architecture (e.g., x86, ARM, POWER) but also on the concrete processor model. To make matters worse, the actual behaviour of the processor is often not precisely described by the processor vendor [Sew+10]. To abstract from the details of particular processor models, *relaxed memory models* are used to describe (often formally) behaviour of given processor architecture. Examples of relaxed memory models of modern processors are the memory model of x86 and x86-64 CPUs described formally as x86-TSO [Sew+10] and the multiple variants of POWER [Sar+11; Mad+12] and ARM [AMT14; Flu+16; Pul+17] memory models.

For the description of a relaxed memory model, it is sufficient to consider operations that affect memory. These operations include loads (reading of data from memory to a register in the processor), stores (writing of data from a register to memory), memory barriers (used to prevent reordering), and *atomic compound operations* (read-modify-write operations and compare-and-swap operation). Compound non-atomic instruction exist on some architectures (e.g., the `add` instruction in x86 can have one memory operand), but they might be rewritten to an equivalent sequence of a load to register, a register modification, and a store, and therefore need not be considered.

2.2.3 The x86-TSO Memory Model⁵

The x86-TSO memory model is a formal description of the memory model used in x86 and x86-64 processors (manufactured by both Intel and AMD) [Sew+10]. It is one of the strongest relaxed memory models

[Sew+10] Sewell et al., “X86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors”.

[Sar+11] Sarkar et al., “Understanding POWER Multiprocessors”.

[Mad+12] Mador-Haim et al., “An Axiomatic Memory Model for POWER Multiprocessors”.

[AMT14] Alglave et al., “Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory”.

[Flu+16] Flur et al., “Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA”.

[Pul+17] Pulte et al., “Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8”.

⁵ This subsection contains text and figures originally from [ŠB18].

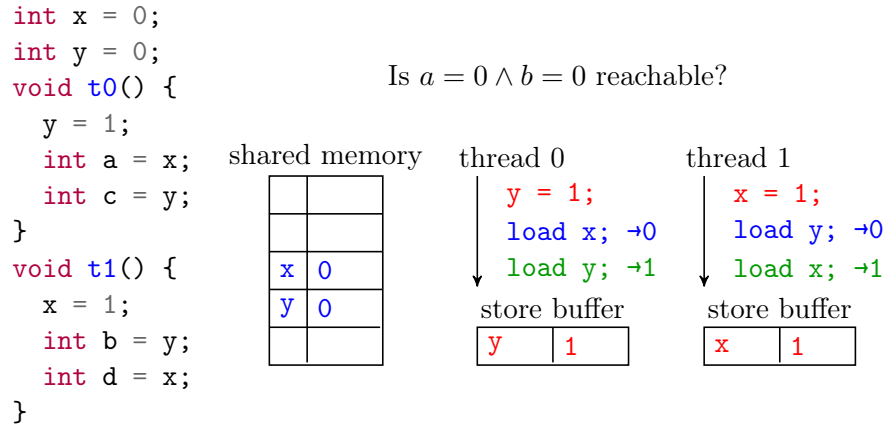


Figure 2.1: A demonstration of the x86-TSO memory model. The thread 0 stores 1 to variable y and then loads variables x and y . The thread 1 stores 1 to x and then loads y and x . Intuitively, we would expect it to be impossible for $a = 0$ and $b = 0$ to both be true at the end of the execution, as there is no interleaving of thread actions which would produce such a result. However, under x86-TSO, the stores are cached in the store buffers (marked red). A load consults only shared memory and the store buffer of the given thread, which means it can load data from memory and ignore newer values from the other thread (blue). Therefore a and b will contain old values from memory. On the other hand, c and d will contain local values from the store buffers (locally read values are marked green). The figure depicts the state of memory and buffers after the code of both threads executed, but before the data was propagated from store buffers to main memory.

– it is relaxed compared to sequential consistency, but not nearly as relaxed as some of the other common memory models such as memory models of ARM and POWER processors. The x86-TSO is very similar to the SPARC Total Store Order (TSO) memory model [SPA94]. It does not reorder stores with each other, and it also does not reorder loads with other loads. The only relaxation allowed by x86-TSO is that a store can appear to be executed later than an independent load that succeeds it in a thread. The memory model does not give any limit on how long a store can be delayed. An example of non-intuitive execution of a simple program under x86-TSO can be found in Figure 2.1.

The operational semantics of x86-TSO is described in [Sew+10]. The corresponding machine has multiple hardware threads (or cores), each with associated local store buffer, a shared memory subsystem, and a shared memory lock. Store buffers are first-in-first-out caches into which store entries are saved before they are propagated to shared memory. Load instructions first attempt to read from the store buffer of the given thread, and only if they are not successful, they read from shared memory. Store instructions push a new entry to the local store buffer. Entries in the store buffer are not visible to threads other than the one owning the store buffer. Atomic instructions include various read-modify-write instructions, e.g. atomic arithmetic operations (that take memory address and a constant),⁶ or compare-and-swap instruction.⁷

[SPA94] SPARC International, “The SPARC Architecture Manual”.

[Sew+10] Sewell et al., “X86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors”.

⁶ These instructions have the `lock` prefix in the assembly, for example, `lock xadd` for atomic addition.

⁷ `lock cmpxchg`

All atomic instructions use the shared memory lock so that only one such instruction can be executed at a given time, regardless of the number of hardware threads in the machine. Furthermore, atomic instructions flush the store buffer of their thread before they release the lock. Therefore, the effects of atomic operations are immediately visible, i.e., atomics are sequentially consistent on x86-TSO. On top of these instructions, x86-TSO has a full memory barrier (`mfence`) which flushes the store buffer of the thread that executed it.⁸

If a programmer (or a compiler) wishes to recover sequential consistency on x86, they need to ensure memory stores are propagated to main memory before subsequent loads execute. This is most commonly done by inserting a memory fence after each store. An alternative approach would be to store using atomic exchange instruction (`lock xchg`) as it can atomically swap value between a register and a memory slot.

One of the specifics of x86 is that it can handle unaligned memory operations.⁹ While the x86-TSO paper does not give any specifics about handling unaligned and mixed memory operations¹⁰, it seems from our experiments that such operations are not only fully supported, but they are also correctly synchronised if atomic instructions are used. This behaviour is in agreement with the aforementioned operational semantics of x86-TSO in which all the atomic operations share a single global lock.

2.2.4 Other Hardware Memory Models

Older works on the analysis of programs under relaxed memory models usually consider the SPARC TSO, PSO and RMO memory models [SPA94] or the memory model of ALPHA [McK10] processors. However, these memory models are not very relevant any more, except for TSO, which is sometimes used interchangeably with the x86-TSO memory model, as they are very similar (the difference is in the behaviour of atomic compound operations, but some works use TSO to stand for the memory model of x86 processors).

Currently, apart from the x86-TSO memory model, mostly the memory models of variants of POWER and ARM processors are relevant. While these processors differ significantly in the area in which they are used (POWER is used in high-performance servers while ARM is mostly used in mobile devices), their memory models share some basic features. Recently, there is also interest in the memory model of RISC-V architecture, which is also similar to memory models of POWER and ARM. All of these types of processors exhibit more relaxed behaviour than x86-TSO, for example, it is possible to reorder a load after a store or to reorder stores or loads with one another (provided they are independent). On POWER it can also happen that a sequence of operations executed by a single thread will be observed by different other threads in different orders. An example of such behaviour can be seen in Figure 2.2.

There are several version (or generations) of ARM and POWER processors, and also multiple versions of memory models. The memory model of ARMv7 is described in [AMT14]. The memory model of newer

⁸ There are two more fence instructions in the x86 instruction set, but according to [Sew+10] they are not relevant to normal program execution.

⁹ Other architectures, for example, ARM require loaded values to be aligned, usually so that the address is divisible by the value size.

¹⁰ E.g., writing a 64-bit value and then reading a 16-bit value from inside it.

[SPA94] SPARC International, “The SPARC Architecture Manual”.

[McK10] McKenney, “Memory barriers: a hardware view for software hackers”.

[AMT14] Alglave et al., “Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory”.

```

int x = 0, y = 0;

void wrt0()    void wrt1()    void read0()  void read1()
{              {              {              {
    x = 1;      y = 1;        int x0 = x;   int y1 = y;
}              }              int y0 = y;   int x1 = x;
}              }              }              }

```

Is it possible that $x_0 = 1 \wedge y_0 = 0 \wedge x_1 = 0 \wedge y_1 = 1$?

Figure 2.2: An independent readers of independent writers example commonly used to demonstrate some of the features of POWER memory models. We assume each of the functions is executed in a separate thread. Here, we are asking if it can happen that while the `read0` observes the new value of x and old value of y the `read1` will observe the old value of x and new value of y . Such an observation would imply that the two modifications are visible to the two readers in different order. This behaviour can indeed be observed on POWER [Sar+11], but not on ARMv8 [Pul+17].

[Flu+16] Flur et al., “Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA”.

[Pul+17] Pulte et al., “Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8”.

[Pul+19] Pulte et al., “Promising-ARM/RISC-V: A Simpler and Faster Operational Concurrency Model”.

[Sar+11] Sarkar et al., “Understanding POWER Multiprocessors”.

[Sar+12] Sarkar et al., “Synchronising C/C++ and POWER”.

[Gra+15] Gray et al., “An Integrated Concurrency and Core-ISA Architectural Envelope Definition, and Test Oracle, for IBM POWER Multiprocessors”.

[Mad+12] Mador-Haim et al., “An Axiomatic Memory Model for POWER Multiprocessors”.

ARMv8 is described in [Flu+16], but it was later revised and simplified in collaboration with ARM and formalised in [Pul+17] and the ARMv8 architecture description. Later in [Pul+19], an operation model of ARMv8 concurrency equivalent with the one presented in [Pul+17] and optimised for program analysis was presented. Interestingly, [Pul+19] also presents a memory model of RISC-V architecture which is similar to the revised ARMv8 memory model.

The basis of POWER memory model was described as an abstract machine in [Sar+11] and later extended in [Sar+12] to cover atomic compound operations. A more detailed memory model of POWER is presented in [Gra+15], including for example behaviour of mixed-size memory operations. Axiomatic description of POWER is provided by [Mad+12] and [AMSS10]. In [Flu+17], the authors describe the behaviour of POWER and ARMv8 in the presence of mixed-size memory operations.

2.2.5 Memory Models of Programming Languages¹¹

Not all programming languages have concurrency and memory model defined in the language. Indeed, C and C++ before the 2011 revisions of their standards did not define concurrency and therefore, the majority of concurrent C code relies on non-standard concurrency. In the absence of programming language support, the programmers wishing to use concurrency have to rely on a combination of library and compiler support that provides (platform-specific) means to support concurrency. For example, on Linux and most POSIX-compatible operating systems, the POSIX threads library (`pthread`) defines ways to launch threads, wait for them, and synchronise their execution using various (blocking) synchronisation primitives such as mutexes and condition variables. A compiler compatible with this library then guarantees that its optimisations will not break this functionality, for example, that it will not reorder opera-

tions around the calls to synchronisation functions. The compiler also usually defines low-level synchronisation primitives which correspond to atomic compound operations. For example, the GCC and clang compilers both define builtin functions such as `__atomic_compare_exchange` and `__atomic_add_fetch`. These builtins allow semi-portable use of atomic operations, in a sense, they are not specific to the concrete hardware platform.¹² Still, they are bound to the given compiler or a group of compilers (such as GCC and clang).

The downside of the library and compiler approach to concurrency is that it makes it hard to write genuinely platform-independent code in the given language. For this reason, many programming languages (eventually) provide concurrency primitives and define a memory model guaranteed by the language.

C and C++ The C++11 [ISO12] and C11 [ISO11] standards introduced support for threading and atomic operations to C++ and C. The memory model of both languages is the same, but they differ in the syntactic ways in which its features are used.¹³

The C++ memory model was revised in the following standards, with the most notable changes in the C++20 [ISO20], including modifications to the strongest sequentially-consistent version of atomic operations. These latest changes mostly remedy problems in interaction between the sequentially consistent operations and weaker memory orderings presented in [Lah+17]. The actual changes are described in [Boe+18].

The C++11 memory model is not formalised in the C++11 standard. An attempt to formalise it was given in [Bat+11], formalising the N3092 draft of the standard [ISO10]. While this formalisation precedes the final C++11 standard, it seems that there were no changes in the specification of atomic operations after N3092. Nevertheless, there are some differences between the formalisation and N3092 (which are justified in the paper). The formalisation was later revised in [Lah+17], which led to the aforementioned revision of concurrency in C++20.

Overall, the C++ memory model is complex and complicated by its intention to allow high-performance on various existing or hypothetical future hardware platforms. Atomic variables and operations play a central role in the C++ memory model. Atomic variables are variables of particular types that define atomic operations such as loads, stores, atomic read-modify-write, and compare-exchange. For an atomic operation, it is possible to specify the required memory ordering: C/C++ allows not only sequentially consistent atomic operations but also weaker (low-level) atomic operations that enable implementation of efficient parallel data structures in a platform-independent way. An example of C++ code that leverages atomic variables is shown in Figure 2.3 and Figure 2.4.

A notable feature of the C++ memory model is that any program that contains a data race on a non-atomic variable¹⁴ has undefined behaviour. Therefore, synchronisation is possible only by atomic variables and concurrency primitives such as mutexes and condition variables.

[AMSS10] Alglave et al., “Fences in Weak Memory Models”.

[Flu+17] Flur et al., “Mixed-Size Concurrency: ARM, POWER, C/C++11, and SC”.

¹¹ *The paragraphs describing memory models of C, C++, Java, and LLVM in this subsection are based on text originally from [Šti18].*

¹² As would be the case if the inline assembly was used to invoke the atomic instruction directly.

[ISO12] ISO C++ Standards Committee, “Standard for Programming Language C++. Working Draft N3337”.

[ISO11] ISO C Standards Committee, “ISO/IEC 9899:201x Committee Draft N1570”.

¹³ For example, C++ defines `atomic` template class for atomic variables, but C relies on the `_Atomic` keyword, as it has no support for (templated) classes. The C threading API is mostly similar to the API of POSIX threads, while the C++ threading has a more modern API which uses classes.

[ISO20] ISO C++ Standards Committee, “C++ Draft International Standard – N4860”.

[Lah+17] Lahav et al., “Repairing Sequential Consistency in C/C++11”.

[Boe+18] Boehm et al., “P0668R4: Revising the C++ memory model”.

[Bat+11] Batty et al., “Mathematizing C++ Concurrency”.

```

std::atomic<int> x;
int y;

void thr0() {
    x.fetch_add(1); // OK
    y++; // RACE
}

void thr1() {
    x++; // also OK
    y++; // RACE
}

```

Figure 2.3: A basic example of C++ atomics. The variable `x` is atomic, and therefore can be safely used in concurrent settings (the standard modification operators for integral types are available and atomic). On the other hand, the variable `y` is not atomic and therefore incrementing it from two threads causes data race.

[ISO10] ISO C++ Standards Committee, “C++ International Standard – N3092”.

¹⁴ Data race is defined as two accesses to the same non-atomic variable, at least one of them being a write, that are not synchronised so that they cannot happen concurrently.

[MPA05] Manson et al., “The Java Memory Model”.

Java The Java memory model is rather different from the C/C++11 one. Its primary goal is to ensure that programs that cannot observe data races under sequential consistency will execute as if running under sequential consistency (the data race free guarantee) [MPA05]. The primary means of synchronisation in Java are mutexes (called monitors in Java), synchronised sections of code (which use monitors internally), and volatile variables, which roughly correspond to sequentially consistent atomics in C++11.

Furthermore, as Java strives to be memory safe, it also defines the behaviour of programs with data races. This behaviour is rather peculiar, as it is primarily concerned with prohibiting out-of-thin-air values – values which, informally speaking, depend cyclically on themselves. These values are primarily prohibited to avoid forging pointers to invalid memory or memory which should be otherwise inaccessible to a given thread [MPA05].

LLVM The LLVM Intermediate Representation has a memory model derived from the C++ memory model, with the difference that it lacks release-consume ordering and offers additional *unordered* ordering that does not guarantee atomicity but makes results of data races defined [LLVM20]. The *unordered* operations are intended to match the semantics of the Java memory model for shared variables.

[LLVM20] LLVM

Project, “LLVM Language Reference Manual”.

¹⁵ This can be generalised to multiple cooperating programs, but the distinction is mostly technical. With multiple programs, message passing is used more often, especially in distributed computing.

¹⁶ And by extension to other general-purpose programming languages with concurrency support, e.g., Java and C#.

2.2.6 Approaches to Multi-Threaded Software

At the highest level, multiple threads of a program can either use *message passing*, or *shared memory*, to communicate.¹⁵ While message passing can be done in any reasonably expressive programming language with support for concurrency, there are programming languages which adopt it as the primary way of implementing concurrent or distributed programs, for example, Erlang or Go. In this work, we are focusing on C++¹⁶ which has no built-in support for message-passing concurrency, therefore we will focus on shared memory concurrency. In C++ and similar languages, message passing can be implemented as an abstraction over shared memory.

```

Data data;
std::atomic< bool > ready;

void thr0() {
    data.load_data();
    read.store( true, std::memory_order::release );
}

void thr1() {
    while ( !read.load( std::memory_order::relaxed ) )
    { }
    std::atomic_thread_fence( std::memory_order::acquire );
    data.do_work();
}

```

Figure 2.4: A simple example of use of low-level atomic API. Here `thr0` loads `data` somehow and then signals to `thr1` that it should be processed. The `data` variable itself is not atomic, and presumably, the type is not made for concurrent access. An atomic boolean is used to wait for `data` in `thr1` – once `data` is loaded the `ready` flag is set, using the *release* memory ordering. The actual waiting in `thr1` uses the weakest *relaxed* ordering which does not cause any synchronisation (it just ensures concurrent changes to the variable itself are consistent), but once the wait ends a fence is executed. The fence uses *acquire* ordering and therefore completes *release-acquire* ordering between `thr0` and `thr1` – anything written by `thr0` before the *release* store is available to `thr1` after the *acquire* fence. An interesting property of this example is that it compiles with no extra synchronisation or atomic instructions on x86 (atomic only affects compiler optimisations there), but uses synchronisation on more relaxed platforms.

Lock-Based Synchronisation Synchronisation plays a crucial role in shared memory concurrency. Concurrent unsynchronised access of two or more threads to the same memory location, with at least one of the threads writing to it, will cause data race which can lead to data corruption and program malfunction. Lock-based synchronisation (critical sections) is often used as it is reasonably simple to understand it, and is often natively supported by programming languages.¹⁷ Furthermore, locks usually use operating system primitives to block waiting threads while another thread executes the critical section, which can increase the performance of systems with more threads than available processors as the waiting threads are inactive and other threads can run in the meanwhile.

On the other hand, synchronisation can degrade the performance of concurrent software. In the extreme case when all interesting work is done in a single critical section executed by many threads, there will be no gain from parallel execution and the overhead of threads and their synchronisation will make the program slower than its sequential version. For this reason, critical sections should be as short as safely possible and different areas of shared memory should be protected by

¹⁷ In C/C++ since the 2011 standards.

different locks so that the program can access different areas of shared memory concurrently. This inherently increases the complexity of synchronisation and risks of data races or deadlocks – indeed deadlocks are easy to avoid with one lock, but with multiple locks, care must be taken if more than one lock is held at one time, which is a common situation.

Atomic Access and Lock-Free Programming To further improve performance, it is sometimes useful to avoid operating-system-assisted synchronisation and use atomic operations instead. These operations can be used to guarantee synchronisation over a word-sized location of memory.¹⁸ Atomic operations are used to implement *lock-free* algorithms and data structures which do not require critical sections at all. In this case, extra care must be taken to the ordering of operations that cannot be performed atomically due to the size limit of atomic instructions and the programmer must carefully evaluate the impact of memory model on their algorithm. An example of (a fragment of) lock-free data structure can be found in Figure 2.5.

Overall, there is often a steep compromise between code simplicity and performance. A code that uses a few locks can often be reasonably understandable, and the programmer need not concern themselves with relaxed memory as correct use of locks makes its presence invisible, but performance might be degraded by synchronisation. On the other hand, lock-free algorithms and data structures can have excellent performance, but designing them and checking their correctness requires a lot of effort and relaxed memory must be taken into account.

2.3 Programming Languages Used in this Thesis

In this work, we are mostly focusing on the analysis of C++. For this reason, we will now introduce some of the features of this programming language relevant to our work. After that, we will introduce LLVM intermediate representation, which is a low-level programming language used by clang and other LLVM-based compilers. DIVINE, the model checker in which the work presented in this thesis is implemented, actually analyses LLVM and uses the clang compiler to translate C++ to LLVM before the analysis.

2.3.1 C++

C++ is a high-level language well suited for a wide variety of projects, from code which directly interacts with hardware to GUI applications. It can be high-performance and has good support for building of abstractions. Since the C++11 version of the C++ standard, C++ also has native support for threads and atomic variables with varying levels of atomicity guarantees. Unless explicitly stated otherwise, all C++ examples in this work use the C++20 standard, as defined by its latest working draft N4860 [ISO20].¹⁹

¹⁸ The size of the atomically-accessed area is usually the same as the size of a pointer on the given platform (e.g., 8 bytes on 64-bit platforms), or sometimes twice the size of the pointer (e.g. 16 bytes on newer x86-64 processors).

[ISO20] ISO C++ Standards Committee, “C++ Draft International Standard – N4860”.

¹⁹ The N4860 draft should represent the final version of the standard. It was submitted to publication, but not yet published at the time of writing of this paragraph.

```

1 void push( const T &x ) {
2     Node *t;
3     Node *ltail = tail.load( std::memory_order_acquire );
4     if ( ltail->write.load( std::memory_order_relaxed )
5         == ltail->buffer + NodeSize )
6         t = new Node();
7     else
8         t = ltail;
9
10    *t->write.load( std::memory_order_relaxed ) = x;
11    t->write.fetch_add( 1, std::memory_order_release );
12
13    if ( ltail != t ) {
14        ltail->next.store( t, std::memory_order_release );
15        tail.store( t, std::memory_order_release );
16    }
17 }

```

Figure 2.5: A push (enqueue) operation of a single-producer-single-consumer lock-free queue from an older version of DIVINE (modified to use C++11 atomic operations). The queue uses a linked list of blocks (of type `Node`). Each block can hold a fixed number of elements of some type `T`. On lines 2–8 the function checks whether there is a room in the last allocated block and allocates a new block if necessary. Both the `tail` and `write` fields in the `Node` are atomic variables, and we explicitly use memory ordering to avoid expensive synchronisation unnecessary on x86-64 processors. Then, on line 10, the actual value is written to the node, on the location pointed to by the `write` field – note that the `write` pointer is atomic, but the value it points to is not atomic. On line 11, we shift the write pointer, which makes the data available to the consumer thread. At this point, we need to use release memory ordering to ensure safety on platforms which are more relaxed than x86 and to prevent reordering by the compiler. The release ordering ensures that any changes performed by the producer thread so far will be available to the consumer once it performs an acquire load on the `write` pointer (to check that the queue is not empty).

Finally, on lines 13–16 we append the linked list of nodes if a new node was created on line 6. Again, we use the release memory ordering to ensure all operations performed so far (in particular the shift of `write` on line 11) are visible once these changes become visible.

Extending this approach to multiple producers is not trivial as the check if there is space available and the publication of the written value would need to be performed at the same time. This would require either modification of two different locations in one atomic step (which is not possible on most processors and there is no C++ API for this operation) or a complete redesign of the queue block.

²⁰ `std::` is a namespace which indicates this type belongs to the C++ standard library. We will sometimes omit it in the examples.

²¹ In the following example, you can see a code that spawns function `worker` in a separate thread and later waits for it.

```
void worker(int a,
           int b) {
    /* do something */
}

int main() {
    std::thread
        w(worker, 1, 2);
    /* do something */
    w.join(); // wait
}
```

²² They are usually based on operating system level blocking primitives – i.e., a thread waiting on these synchronisation primitives is suspended and not using any resources.

Threads In C++ a thread is started by creation of an object of type `std::thread`.²⁰ Later, the thread can be waited-for by calling the `join` method of the `std::thread` object. Joining will block until the thread we are joining finishes.²¹

High-Level Synchronisation primitives C++ comes with various high-level synchronisation primitives.²² Two of the primary synchronisation primitives are *mutexes* and *condition variables*. Mutexes can be used to create mutual exclusion: only one thread can lock a given mutex at any point. Condition variables allow some threads to wait for a signal from other threads. They are often used in producer-consumer scenarios to signal to consumers that there is some data ready for them. They have two main operations – `wait` and `signal` – `wait` is a blocking call that suspends its calling thread until another thread calls `signal`. Waiting can only be ended if `signal` is called after the wait started, the conditional variable has no way to detect that `signal` was called before `wait`. Furthermore, due to the limitations of some platforms, `wait` is allowed to end spuriously (without being signalled). For these two reasons, condition variables are usually used together with a shared variable which indicates whether or not the consumer thread should proceed; this variable should be guarded by the same mutex the condition variable uses for signalling.

Atomic Variables and Low-Level Synchronisation C++ has also support for atomic variables and atomic operations with them. These atomic variables allow the program to take advantage of atomic hardware instructions available on most platforms. Atomic variables are mainly used in lock-free data structures and algorithms.

Atomic variables in C++ are variables of one of the atomic types: `std::atomic_flag` and `std::atomic<T>` for some type `T` (which is usually an integral or pointer type, for example `std::atomic<int>`). For integral types, `std::atomic<T>` defines various operations that allow atomic modification of the value of an atomic variable: for example, by calling `fetch_add` on an atomic variable it is possible to atomically increment its value and return its original value before the increment. Some of these operations are also available with the operator syntax (e.g., using the operator `+=`). For all types, it is possible to exchange the current value with a new one (returning the previous value) and to perform compare-exchange (also sometimes called compare-and-swap), which is a compound operation that atomically checks that the value of an atomic variable is equal to a specified value and if so replaces the original value with a new one.

The atomic operations are not required to be lock-free but can use locks internally (with the exception of `std::atomic_flag`). Therefore, atomic types can be implemented even for data types larger than the platform-specific limit of atomic instructions.

²³ There is a single global ordering of atomic operations on which all threads agree.

Memory Ordering for Atomic Operations Without any additional settings, all atomic instructions in C++ are sequentially consistent.²³ However, C++ aims to allow programmers to utilise the

performance of a given platform fully, and therefore, it is possible to specify weaker constraints for atomic operations. These constraints are specified using a *memory order*. We will now informally describe available memory orders.

`std::memory_order::seq_cst` is the strongest and default memory order. It forces operations with this memory order to be sequentially consistent. It also prevents reordering of other operations (atomic or not) around a sequentially consistent operation.

`std::memory_order::release` is memory order used with store operations. It prevents previous memory operations in the same thread to be reordered after the release store.

`std::memory_order::acquire` is used with load operations. It prevents later memory operations in the same thread to be reordered before the acquire load. A release store and a subsequent acquire load from the same variable create a synchronisation that ensures that all modifications performed in the storing thread before the release store are visible to the loading thread after the corresponding acquire load.

`std::memory_order::acq_rel` is used with atomic compound operations and combines the acquire and release orderings.

`std::memory_order::consume` is a weaker form of `acquire` that only affects data-dependent variables. This memory ordering is not widely used and is often treated as `acquire` by compilers.

`std::memory_order::relaxed` is the weakest atomic ordering. It guarantees no synchronisation with operations on other memory locations. It only guarantees atomicity of operations over the given location. Relaxed ordering can be used, for example, to implement atomic counters used for statistics.

Exceptions C++ has support for exceptions and is also able to specify that a given function is not allowed to throw an exception. In C++, it is possible to throw value of any type as an exception. When an exception is thrown, it propagates to callers of the function which have thrown it until it triggers a `catch` code block that can catch it – a `catch` block that either catches exactly the type of the exception or if the class of the exception uses inheritance, the exception can also be caught by a `catch` block that catches some of the predecessor types of the exception. Whenever the propagation of an exception causes a scope of some variable to end, the variable’s destructor is called to release resources associated with it (for example, close an open file, release memory or release a mutex).

If a function is marked as `noexcept` and an exception would propagate from it, the program terminates.

Deterministic Destruction of Objects One of the core features of C++ is that it has deterministic destruction of objects, i.e., the

end of the lifetime of objects is precisely known (as opposed to being determined by the garbage collector in garbage-collected languages such as Java or C#). In C++, the end of life of an object happens either when it goes out of scope (for scope-allocated objects), when it is explicitly deallocated (for dynamically allocated objects) or when the program terminates (for global objects). This allows C++ destructors, which are special functions executed at the end of life of an object, to be used for resource control. For example, an `std::unique_lock` object might be initialised with an instance of a mutex, and it will automatically lock the mutex when the `unique_lock` is constructed and release the mutex when it goes out of scope. The same is used for example in C++ file streams to close the file handle when the stream goes out of scope, or for memory management.

An important property of this approach to resource management is that it is exception-safe. If the resource-owning object is a local variable, the resource will be automatically released at the end of its scope, whether or not the end of scope was caused by exception propagation or normal control flow.

2.3.2 LLVM IR

LLVM is a compilation infrastructure which can be used to build optimising compilers. A compiler built on LLVM consists of a language-specific frontend that processes the source code and produces a language-independent LLVM intermediate representation, an optimiser that runs on the LLVM intermediate representation, and a code generator that produces executable code for the given platform. LLVM intermediate representation (LLVM IR, LLVM code, or just LLVM), is a low-level programming language mostly independent of both the high-level language of the original program and the assembly language of the given hardware platform. Nevertheless, LLVM IR is somewhat influenced by the languages that are mainly translated to it, namely C and C++.

LLVM IR is a type-safe assembly-like language. Its basic operations are *instructions* which take inputs of a specific type and produce an output of possibly different type. Values can be stored either in registers (each register is only assigned at one place in the code – the code is in static single assignment (SSA) form) or in memory. Memory can be further divided into global variables, which exist for the entire run of the program, and dynamically allocated memory, which is obtained by a call to an allocation function provided by the platform. The allocation function is expected to be externally provided; memory allocation is not a part of LLVM.

Memory Manipulation in LLVM Unlike the x86 machine code, most LLVM instructions do not modify memory directly but work with registers only. Therefore, to change a value in memory, it is first necessary to load it, using the `load` instruction, then modify it, and finally store it using the `store` instruction. Two more instructions can access memory, and these are used for *atomic compound operations*. The `atomicrmw` instruction (atomic read-modify-write) can atomically perform a load, arithmetic or logic operation, and a store or atomically

replace a value in memory with another value. In all cases, it returns the old value of the memory location. The `cmpxchg` (compare exchange) can atomically check if the value in memory is the same as expected, and if so, replace it with a new value. Atomic compound operations have memory order argument which specifies their level of atomicity. Similarly, `load` and `store` instructions can be atomic, and their atomic versions also come with a memory order argument. Memory orders in LLVM are based on memory orders in C++ (see Section 2.3.1).

Threads and High-Level Synchronisation in LLVM LLVM has no primitives for starting and handling threads nor for high-level synchronisation of threads in shared memory (mutexes, condition variables, ...). Therefore, this functionality has to be provided by libraries, which matches well with the programming languages often translated to LLVM, which usually implement threading using a library of thread-manipulation and synchronisation primitives. Nevertheless, LLVM has a notion of thread-local variables; i.e., variables that exist in a separate copy in each thread.

Memory Model The memory model of LLVM is mostly based on the memory model of C++. However, instead of atomic variables, it uses only atomic operations (i.e., it is theoretically possible to combine atomic and non-atomic access to the same variable in LLVM).

Exceptions LLVM has support for exceptions. Namely, there are two ways in which a function can be called in LLVM. The `call` instruction is used for calls which either cannot throw an exception, or which can throw an exception, but the exception does not need to be inspected or caught in the functions that perform the `call`. The `invoke` instruction is used if the exception needs to be intercepted. An `invoke` is a branching point. If the function called by `invoke` returns normally, the `invoke` behaves like a `call` followed by a jump to the next instruction after the `invoke`. If an exception is propagated through `invoke`, it transfers control to a block of code which starts with a `landingpad` instruction. The `landingpad` instruction returns information about the exception and the code which starts with it then decides how to handle the exception. The exception can be handled by this function, or cleanup can be run, and the propagation of the exception can be resumed using the `resume` instruction.

Interestingly, there is no instruction in LLVM to throw an exception. Instead, the whole design assumes the platform for which the code is compiled provides an exception support library – so-called *unwinder* which can throw the exception. When the program is executing, the unwinder also takes care of the actual propagation of the exception through the call stack (*unwinding*). For this purpose, it needs metadata which is generated by the code generator from the information in the `invoke`, `landingpad`, and `resume` instructions.

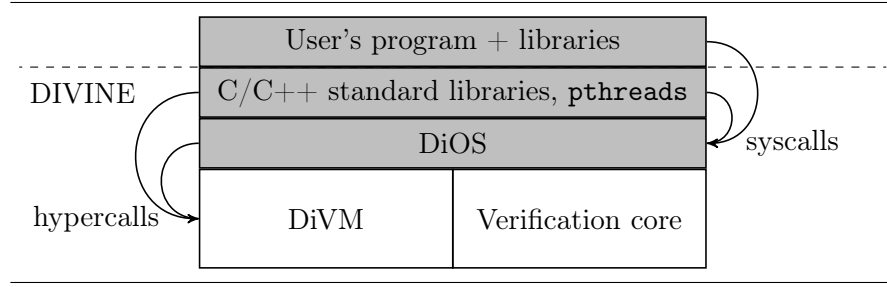


Figure 2.6: Overview of the architecture of DIVINE. The shaded part consists of LLVM code which is interpreted by DiVM. *This figure was originally created for [Bar+17].*

2.4 DIVINE

DIVINE is an open-source verifier for C and C++ programs with a focus on concurrency and hard to discover bugs [Bar+17; Roč20]. It aims to be a general tool usable by programmers and can work with a wide variety of programs and check for different property violations, including assertion failures, memory access errors, memory leaks, use of uninitialised memory and mutex locking errors. There is also limited support for liveness properties and an extension to checking nontermination of parallel programs (Chapter 6, [ŠB19]). Furthermore, DIVINE aims at full support of C and C++, including their standard libraries. Currently we support C++17 with most of its standard library (Chapter 4, [ŠRB17]).

DIVINE’s platform model is loosely based on x86-64 Linux – it uses 64 bit pointers and data type sizes and alignments are the same as used on this platform. With the optional support for the x86-TSO memory model (Chapter 5, [ŠB18]), DIVINE also respects the memory models of x86-64 processors. Among the most significant differences between DIVINE and Linux on x86-64 are different calling conventions and stack layout; however, these differences should be transparent to a correct C or C++ program, and programs which rely on a platform-specific layout of the stack will be reported as buggy in most cases (the stack layout is not guaranteed by the C/C++ standards, and therefore such programs are not well-defined).

2.4.1 Architecture of DIVINE

The architecture of DIVINE is illustrated by Figure 2.6. At the base of DIVINE, there is an explicit-state core which controls exploration of the state space. To execute the program, DIVINE uses DiVM, a verification-oriented virtual machine that executes LLVM instructions and DiVM extensions to LLVM IR called *hypercalls* [RŠČB18]. DiVM hypercalls behave as functions from the point of view of LLVM IR, but they are treated as instructions by DiVM. These hypercalls handle operations like memory allocation and freeing and nondeterministic choice. They can also be used to switch stacks (for example to implement threading), associate metadata with addresses, and to control the exploration of the state space using interrupt points and cancellation.

[Bar+17] Baranová et al., “Model Checking of C and C++ with DIVINE 4”.

[Roč20] Ročkai, “DIVINE 4”.

[ŠB19] Štill et al., “Local Nontermination Detection for Parallel C++ Programs”.

[ŠRB17] Štill et al., “Using Off-the-Shelf Exception Support Components in C++ Verification”.

[ŠB18] Štill et al., “Model Checking of C++ Programs Under the x86-TSO Memory Model”.

[RŠČB18] Ročkai et al., “DiVM: Model checking with LLVM and graph memory”.

In DIVINE the DiVM has only limited knowledge of threads – it can take their existence into account for the state-space reductions, but cannot start or terminate them and does not direct their scheduling. Instead, scheduling is controlled by a scheduler linked to the verified program which uses the nondeterministic choice hypercall provided by DiVM to decide which thread to execute. Thread creation and termination is also implemented in a library. Scheduling, thread management and a basic POSIX-compatible interface including a filesystem model are grouped into DiOS, a verification-oriented operating system [Roč+19]. On top of DiOS, there is a set of libraries including implementations of C and C++ standard library, POSIX thread library (`pthread`), and unwinder for exception handling (see also Chapter 4 and [ŠRB17]).

[Roč+19] Ročkal et al., “Reproducible Execution of POSIX Programs with DiOS”.

[ŠRB17] Štill et al., “Using Off-the-Shelf Exception Support Components in C++ Verification”.

2.4.2 Program Compilation & Libraries

All the parts of DIVINE described so far process LLVM IR. To be able to handle C and C++, DIVINE uses the clang compiler (which is part of the LLVM project). However, if DIVINE is to analyse a program, it needs definitions of all the library functions this program uses in the form of LLVM IR (alternatively DiVM itself would need to be able to understand these functions). Therefore, DIVINE needs substantial control over the compilation process; for example, it is not possible to use system-installed (binary) libraries with DIVINE.

To make compilation reasonably simple for users, DIVINE integrates the clang compiler using its library interface. This integrated compiler compiles programs in an environment that contains only the libraries available in DIVINE, automatically links them, and produces LLVM IR that can be directly analysed with DIVINE. The compiler can be invoked either automatically by DIVINE when it is executed with a C or C++ file, or with a standalone compiler that can be used in place of clang.

As we have already mentioned, DIVINE needs LLVM IR of the used libraries. Therefore, DIVINE ships with implementations of standard C and C++ library, the POSIX threads library, and a stack unwinder library. The C and C++ standard libraries are based on existing projects (that aim at normal execution), namely PDCLib for the C library and `libc++` and `libc++abi` for the C++ library.

File System and POSIX API C and C++ standard libraries provide, among others, access to the file system. To be able to use the file system in programs analysed by DIVINE, DIVINE provides two possible approaches. The first is a virtual file system provided by DiOS. In this mode, most file system operations are modelled by our Virtual File System library (which is part of DiOS). The program can work with the file system as usual, but this has no effect on the file system of the computer the program runs on, the file system is part of the program’s state. The file system can either start with an empty snapshot, or it can load a snapshot of a directory at the start.

An alternative approach is to allow the program to access the file system of the machine. However, this is not possible during normal state-space exploration that explores the state space in BFS-like order

[KRB17] Kejstová et al., “From Model Checking to Runtime Verification and Back”.

[Roč+19] Ročkai et al., “Reproducible Execution of POSIX Programs with DiOS”.

[LRB18] Lauko et al., “Symbolic Computation via Program Transformation”.

[CLOR19] Cortesi et al., “String Abstraction for Model Checking of C Programs”.

[ŠRB17] Štill et al., “Using Off-the-Shelf Exception Support Components in C++ Verification”.

[ŠB18] Štill et al., “Model Checking of C++ Programs Under the x86-TSO Memory Model”.

[ŠB19] Štill et al., “Local Nontermination Detection for Parallel C++ Programs”.

[RBB13] Ročkai et al., “Improved State Space Reductions for LTL Model Checking of C & C++ Programs”.

[RŠČB18] Ročkai et al., “DiVM: Model checking with LLVM and graph memory”.

²⁴ In LLVM these are `load`, `store`, `cmpxchg`, and `atomicrmw`.

as it could cause interference between independent parts of the state space. Instead, DIVINE can execute a single run of the program (giving random values to nondeterministic choices), record all supported system calls, and then use this recording in analysis [KRB17]. A limitation of this approach is that it currently works only if all the system calls are executed in the same order in all runs of the program (which is unlikely if the program uses file system from more than one thread).

On top of the file system support, DIVINE also has support for other parts of the POSIX API, including parts of networking and clock API [Roč+19].

2.4.3 Program Instrumentation

To avoid unnecessary complexity in DiVM, DIVINE uses program instrumentation as a preprocessing step. This step runs on the LLVM IR of the program and all the libraries linked to it. Instrumentation adds potential interruption points to the program to specify points at which the scheduler can be invoked. These interruption points are used at memory access instructions and at back-edges of loops (to make sure DiVM does not attempt run an infinite cycle). Furthermore, the instrumentation is used to calculate metadata used by DiOS.

Instrumentation is also used to implement some extensions of DIVINE. It is used by the symbolic and abstract extensions of DIVINE [LRB18; CLOR19], for C++ exceptions (Chapter 4, [ŠRB17]), if the program is analysed for relaxed memory behaviour (Chapter 5, [ŠB18]), and to enable local nontermination checking (Chapter 6, [ŠB19]).

2.4.4 State Space Reductions

DIVINE relies on state space reduction to be able to handle reasonably sized realistic parallel programs [RBB13; RŠČB18]. The basic principle of the reductions used in DIVINE stems from the observation that most of the instructions of one thread do not interfere with instructions of other threads. For this reason, it is sufficient to enable thread interleaving only after a block of instructions, provided we can ensure that only one instruction that can interfere with instructions of other threads can happen in each block and that each block terminates.

In DIVINE, we use heuristics to detect which instructions can interfere with other threads. A simple version of such a heuristic is that only instructions which access memory can interfere with other threads in LLVM.²⁴ For this reason, the instrumentation in DIVINE only inserts potential interrupt points at these instructions. When successors of a state are generated by DiVM, it uses information from the potential interrupt points (which includes memory address range and a type of the access) to determine if the interrupt is necessary. To do this, DIVINE locally searches for actions of other threads that can interfere with the given thread.

As the size of instruction blocks executed together by DiVM is dynamic, it needs to ensure that each block is finite. Therefore, DiVM detects control flow cycles and ends a block of instructions if a back-edge in a control flow graph is repeated.

2.4.5 Implementation and Availability

DIVINE is open-source software written mostly in C++. DIVINE source codes, as well as instruction on how to build and use it, can be found on the project website [Roč20].

[Roč20] Ročkai,
“DIVINE 4”.

Chapter 3

State of the Art

Historically, automatic analysis techniques for parallel programs focused on the analysis of models of systems. A programmer wishing to use such a tool would either start by creating a model of the system (in the specification step of the development), and then provide an executable implementation for this model or, if they already had a working product, they would have to create a model to analyse it. Such tools include for example SPIN [Hol97], older versions of DIVINE [BBČŠ05], LTSmin [Kan+15] and BMC [BCCZ99]. The model-based approach requires an extra investment in the modelling phase and, even if analysis of the model concludes it is correct, it does not prove that the final product is indeed correct.

Later, with improvements of both analysis techniques, as well as overall improvements in available computing power, analysis tools for programs written in mainstream programming languages become available. Early examples of such tools are Java Pathfinder [Vis+03] (an explicit-state model checker for Java) and CBMC [CKL04] (a SAT-based bounded model checker for C). Since 2012, the Software Verification Competition (SV-COMP) [Bey20] aims to showcase tools that support direct verification of software written in C and lately also Java. While it includes mostly sequential programs, there is also a subcategory for parallel C programs in SV-COMP.

We will focus in more details on automatic techniques for verification of parallel programs. We will not consider program analysis techniques which require substantial manual effort (e.g., proof-assistant-based techniques), or techniques which are not applicable to realistic programs (e.g., techniques which use a modelling language). We will also mostly disregard techniques with no support for parallel programs. Finally, we are primarily concerned with techniques that were implemented and evaluated – the existence of a tool that can handle a programming language (as opposed to a modelling language) can be seen as a witness of maturity of a technique.

3.1 Explicit-State Model Checking

Explicit-state model checking is based on an exhaustive exploration of the state-space graph. It checks that a given (finite-state) system satisfies given property. The property is often provided by an LTL

[Hol97] Holzmann, “The model checker SPIN”.

[BBČŠ05] Barnat et al., “DiVinE – The Distributed Verification Environment”.

[Kan+15] Kant et al., “LTSmin: High-Performance Language-Independent Model Checking”.

[BCCZ99] Biere et al., “Symbolic Model Checking without BDDs”.

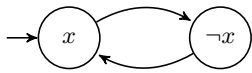
[Vis+03] Visser et al., “Model checking programs”.

[CKL04] Clarke et al., “A Tool for Checking ANSI-C Programs”.

[Bey20] Beyer, “Advances in Automatic Software Verification: SV-COMP 2020”.

[BK08] Baier et al., “Principles of Model Checking”.

¹ A finite state space can contain cyclical infinite behaviour – a loop in the state space.



[Pel93] Peled, “All from one, one for all: on model checking using representatives”.

[God+96] Godefroid et al., “Partial-Order Methods for the Verification of Concurrent Systems”.

[Lip75] Lipton, “Reduction: A Method of Proving Properties of Parallel Programs”.

[YG04] Yorav et al., “Static analysis for state-space reductions preserving temporal logics”.

[BBR12] Barnat et al., “Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs”.

[RBB13] Ročkai et al., “Improved State Space Reductions for LTL Model Checking of C & C++ Programs”.

[RŠČB18] Ročkai et al., “DiVM: Model checking with LLVM and graph memory”.

[Vis+03] Visser et al., “Model checking programs”.

formula and the automata-based approach to LTL verification is used (i.e., the problem is reduced to the problem of repeated reachability of a state in the state-space graph) [BK08, §5.2]. In the particular case of safety properties, it is sufficient to perform graph search for a state which violates the safety property, for example, using the depth-first search algorithm, or any other graph search.

The advantage of explicit-state model checking is that it is conceptually easy to apply it to verification of parallel programs (under sequential consistency), as the interleaving semantics of threads naturally gives rise to the state space graph. Furthermore, explicit-state model checking does not require the program to terminate; it is sufficient that its state space is finite.¹

State Space Explosion & Reduction Techniques In practice, explicit-state model checking is prone to the *state-space explosion* problem: the number of states in the state-space graph of a reasonable system can easily be so big it is not possible to store the state space in available memory and explore the state space in a reasonable time. Since the algorithm which explores the state-space graph needs to detect which states were already seen to ensure termination (and to check for LTL properties), available memory is often the limiting factor, at least without advanced state space reduction techniques.

Several state-space reduction techniques were introduced to mitigate the state-space explosion. Using these techniques, it is possible to explore only some of the states of the state space in such a way that the property holds for these states if and only if it holds for the entire state space. One of these techniques is *Partial Order Reduction* (POR) which can eliminate some states by exploring independent events only in one particular order [Pel93; God+96].

Another wide family of reductions are techniques that can coalesce a path in a state space into a single edge and hide all intermediate states. Lipton introduced an early example of this idea in [Lip75]. He used the notion of right movers (“resource acquire operations”) and left movers (“resource release operations”) to identify statements in a program which can be executed atomically. However, in the analysis of realistic parallel programs, a notion of instruction (or action) visibility is often used. A group of instructions from one thread can be executed atomically, provided that at most one of them is observable by the other threads, and that this grouping does not interfere with checking of the verified property or termination of the search. Both static and dynamic reduction methods were proposed, under many names, including *D-reduction* [Lip75], *path reduction* [YG04], *τ reduction* [BBR12] for the static variants and *$\tau+$ reduction* [RBB13] and [RŠČB18, Section 6] for the dynamic ones. These reductions are also often used without naming the technique, for example in Java Pathfinder [Vis+03]. Quite naturally, the dynamic methods are better suited for the complex control- and data-flow of realistic programs. Both partial order reductions and path reductions are also often used in tools based on other principles than explicit-state exploration.

Additionally, there are variants of *symmetry reduction* that reduce the state space by coalescing states which differ only in properties not relevant for the program analysis [CEJS98]. For example, using the *heap-symmetry* reduction, two states that differ only in the order in which memory objects were allocated can be considered equal [RBB13], [RŠČB18, Chapter 6]. A dead-variable elimination [YG04] can also be seen as an instance of symmetry reduction.

These state-space reduction techniques can reduce the number of states by several orders of magnitude [RBB13], and therefore enable verification of realistic (but still relatively small) programs.

Interestingly, the proposed reduction techniques which aim to preserve verified properties are not always correct, for example the original version of $\tau+$ reduction considered only stores to be visible, but it was later shown that repeated loads have to be also visible (shown by me in [Šti16], fixed in [RŠČB18, Section 6]).² A similar problem was present in [CF11], where ESBMC could perform two conditional jumps that read the same shared variable with no context switch between them (section 3.1, rule R3). Interestingly, the authors notice the possibility of missing context switches but only introduce an option to fix it, leaving the problematic behaviour as default.

To further improve the capabilities of explicit-state model checking, several techniques for memory-efficient representation of the set of visited states were introduced. These techniques include *hash compaction* and *bitstate hashing* [Hol98], which are incomplete techniques that store hashes of states instead of storing the entries states (and therefore can omit some parts of the state space if there is hash collision), *conditional and external storage of states* [HW07], and *lossless compression techniques* [RŠB15; Laa19].

Data Nondeterminism While explicit-state model checking can easily represent control-flow nondeterminism, it is not well suited for data nondeterminism, as it is not practical (or even possible) to explicitly enumerate all possible values of data domains. Therefore, if data nondeterminism is required, explicit-state model checking needs to be combined with some technique for symbolic or abstract data representation [Päs+13; MBLB16].

Tools The pioneering tool is the SPIN LTL model checker [Hol97; Hol04]. SPIN has very limited support for analysis of realistic programs. It targets a parallel modelling language PROMELA, which has support to embed C code to define an atomic step. In [ZJ08] SPIN was extended to have partial support for C. Even so, this version needs to have a test driver in PROMELA.

Java Pathfinder (JPF) [Vis+03; AV19] is an explicit state model checker (with symbolic extensions) for Java and other JVM-based³ languages (e.g., Scala, Kotlin). From its beginning, JPF targets parallel Java programs. It can check for safety properties, namely for uncaught exceptions which in Java also subsume assertion checking and bound checking. To reduce the state space, JPF uses hash-compaction (and therefore under-approximates all possible behaviours), symmetry

[CEJS98] Clarke et al., “Symmetry reductions in model checking”.

[RBB13] Ročkai et al., “Improved State Space Reductions for LTL Model Checking of C & C++ Programs”.

[RŠČB18] Ročkai et al., “DiVM: Model checking with LLVM and graph memory”.

[YG04] Yorav et al., “Static analysis for state-space reductions preserving temporal logics”.

[Šti16] Štill, “LLVM Transformations for Model Checking”.

² Consider a program with threads **t0** and **t1**:

```
int x = 0;
void t0() {
    x = 1;
}
void t1() {
    int a = x;
    assert(a == x);
}
```

If the two loads in **t1** are not considered visible, they are coalesced, and the assertion violation is missed.

[CF11] Cordeiro et al., “Verifying Multi-Threaded Software Using SMT-based Context-Bounded Model Checking”.

[Hol98] Holzmann, “An analysis of bitstate hashing”.

[HW07] Hammer et al., “To Store or Not To Store” Reloaded: Reclaiming Memory on Demand”.

[RŠB15] Ročkai et al., “Techniques for Memory-Efficient Model Checking of C and C++ Code”.

[Laa19] Laarman, “Optimal compression of combinatorial state spaces”.

[Päs+13] Păsăreanu et al., “Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis”.

[MBLB16] Mrázek et al., “SymDIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration”.

[Hol97] Holzmann, “The model checker SPIN”.

[Hol04] Holzmann, “The SPIN model checker: Primer and reference manual”.

[ZJ08] Zaks et al., “Verifying Multi-threaded C Programs with SPIN”.

[Vis+03] Visser et al., “Model checking programs”.

[AV19] Artho et al., “Java Pathfinder at SV-COMP 2019 (Competition Contribution)”.

³ Java is not executing directly on the target hardware but instead uses an intermediate language which is run by JVM.

⁴ Java programs can use the Java Native Interface (JNI) to interfere with native code, for example, to access operating system primitives directly.

[Nol+19] Noller et al., “Symbolic Pathfinder for SV-COMP”.

[KPV03] Khurshid et al., “Generalized Symbolic Execution for Model Checking and Testing”.

[FLP17] Fromherz et al., “Symbolic Arrays in Symbolic Pathfinder”.

[Ban+16] Bang et al., “String Analysis for Side Channels with Segmented Oracles”.

reduction (with respect to class loading order and heap symmetry), optionally predicate abstraction (with user-provided predicates), partial order reduction, and groups instructions with local effects. Interestingly, the instruction grouping in JPF uses a heuristic which can cause it to miss some behaviours. JPF also has models of parts of the Java standard library (including limited IO support) and limited support for execution of native code from the Java code⁴. In addition to scheduling nondeterminism, JPF can use explicit choice as an additional source of nondeterminism. The symbolic extension, Symbolic Pathfinder (SPF) [Päs+13; Nol+19] adds support for symbolic data representation using symbolic execution. It supports test generation and detection of assertions and errors related to parallel execution. It primarily targets unit tests and sub-system level testing and can use unit preconditions and combine symbolic and explicit execution. To explore different possible interleavings, SPF uses the explicit JPF without state comparison and with depth bound to ensure termination (the state comparison can be enabled, but it disregards the symbolic data and therefore can miss many behaviours). SPF can handle some instances of dynamically allocated linked symbolic data by *lazy initialisation* – when a symbolic pointer is accessed, it can be expanded to either null pointer or another node of symbolic data [KPV03]. SPF has support for symbolic arrays [FLP17] and symbolic strings [Ban+16].

An unnamed explicit-state model checker for C# is presented in [HR06]. It targets a subset of .NET bytecode (which is an intermediate representation into which C# and other .NET languages are translated), and it analyses parallel programs running under the .NET relaxed memory model. It can be used to detect behaviour that differs from any possible sequentially consistent behaviour and to insert memory barriers to recover sequential consistency.

DIVINE [Bar+17] is an explicit-state model checker developed by our research group. Historically, it targeted several modelling languages for parallel systems and LTL verification using parallel and distributed algorithms, but later it shifted towards the analysis of C and C++ using the LLVM intermediate representation. While it now aims primarily at verification of safety properties (assertion violations, memory access safety, detection of use of undefined variables, detection of memory leaks, numeric manipulation errors, and deadlock-freedom for the POSIX mutexes), it also has limited support for LTL and an extension for detection of nontermination in parallel programs (Chapter 6, [ŠB19]). To tackle realistic programs, DIVINE uses a dynamic detection of invisible actions (τ + reduction) and an efficient representation of program memory which facilitates heap-symmetry reduction and state-space compression [RŠČB18]. DIVINE also supports symbolic and abstract data representation using program transformations [LRB18]. Currently, it supports symbolic bitvector manipulations for both integral and floating-point data types and symbolic string representation [CLOR19]. With the symbolic data representation using bitvectors, DIVINE uses SMT solvers to check for feasibility of traces and to compare symbolic states – this way it retains the ability to join states that are semantically equivalent even if the symbolic data are represented by a different

formula. The state comparison allows DIVINE with symbolic data to ensure termination for programs with finite symbolic state space, and it also enables checking LTL properties. One of the main goals of DIVINE is to support verification of C and C++ programs which use existing libraries. To this end, DIVINE has almost complete standard C and C++ libraries (as of C++17), the POSIX thread library (`pthread`), and it also supports C++ exceptions (Chapter 4, [ŠRB17]). To reflect the behaviour of parallel programs on contemporary hardware, DIVINE has support to analyse programs with respect to the x86-TSO memory model (Chapter 5, [ŠB18]). DIVINE also has an in-built compiler based on LLVM’s `clang` and supports significant parts of POSIX file system and process APIs [Roč+19].

SymDIVINE [MBLB16] was another explicit-state model checker from our research group. It combines explicit control flow handling with a symbolic representation of data using bitvectors (with symbolic state equality). SymDIVINE targets safety and LTL properties in C programs. This tool is now discontinued in favour of the aforementioned symbolic data support in DIVINE.

3.2 Stateless Model Checking

Compared to explicit-state model checking, *stateless model checking* (SMC) avoids storing the set of visited states and therefore has decreased memory consumption. Furthermore, since the state representation is not required to be as compact as possible, a stateless model checker can have a simpler representation of states. Stateless model checking was introduced in [God97], and it aims at safety analysis of terminating realistic parallel programs. A stateless model checker usually explores the state space in a depth-first manner, and it can explore some parts of the state space multiple times (since it does not store the set of visited states). Therefore, the requirement that the input program terminates is necessary to ensure the analysis terminates. In practice, this requirement is often ensured by imposing loop iteration bounds. If the program under test requires more iterations of a loop, the loop bound can be increased, or the analysis can be terminated as inconclusive.

Stateless model checking is also sometimes presented under names like *systematic concurrency testing* [CGS13].

State Space Reductions Without additional state space reductions, SMC would lead to redundant explorations of many parts of the state space. Indeed, in parallel programs, it is common that two or more actions of different threads are independent, and regardless of their order, they lead to the same end state. In this case, a stateless model checker would explore a state as many times as is the number of paths from the initial state to this state (in the worst case the number of paths to a given state can be exponential to its distance from the initial state). To mitigate this problem, *dynamic partial order reduction* (DPOR) [FG05] is often employed with SMC. DPOR is a version of partial order reduction that tightly integrates with the SMC exploration algorithm and keeps track of parts of the state space which still need

[HR06] Huynh et al., “A Memory Model Sensitive Checker for C#”.

[Bar+17] Baranová et al., “Model Checking of C and C++ with DIVINE 4”.

[ŠB19] Štill et al., “Local Nontermination Detection for Parallel C++ Programs”.

[RŠČB18] Ročkait et al., “DiVM: Model checking with LLVM and graph memory”.

[LRB18] Lauko et al., “Symbolic Computation via Program Transformation”.

[CLOR19] Cortesi et al., “String Abstraction for Model Checking of C Programs”.

[ŠRB17] Štill et al., “Using Off-the-Shelf Exception Support Components in C++ Verification”.

[ŠB18] Štill et al., “Model Checking of C++ Programs Under the x86-TSO Memory Model”.

[Roč+19] Ročkait et al., “Reproducible Execution of POSIX Programs with DiOS”.

[MBLB16] Mrázek et al., “SymDIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration”.

[God97] Godefroid, “Model Checking for Programming Languages Using VeriSoft”.

[CGS13] Christakis et al., “Systematic Testing for Detecting Concurrency Errors in Erlang Programs”.

[FG05] Flanagan et al., “Dynamic Partial-order Reduction for Model Checking Software”.

to be explored. Using DPOR, SMC can avoid redundant exploration of equivalent paths in the state space.

Many works are concerned with the design of efficient DPOR methods both for parallel programs running under sequential consistency (interleaving semantics) and for various relaxed memory models. It is usually accomplished by a combination of two aspects: an equivalence of traces and an exploration algorithm which ensures at least one trace (and in optimal case exactly one trace) from each equivalence class is explored. The trace equivalence has to be designed in such a way that for each of its classes either all traces contain only safe states, or all traces contain an unsafe state – i.e., the equivalence preserves safety properties.

Over the years, multiple DPOR techniques were introduced [FG05; SA07; Tas+12; SKH12; AAJS14; ZKW15; Cha+17; AAJN18; AJLS18; Abd+19]. Among these techniques, [AAJS14] is interesting since it provides an optimal algorithm for equivalence based on *Mazurkiewicz traces* [Maz87] (where traces are considered equivalent if one of them can be obtained from the other by swapping adjacent non-conflicting execution steps). The optimal DPOR presented by [AAJS14] is optimal for the given trace equivalence (i.e., it explores exactly one execution in each equivalence class of the used trace equivalence). In recent years, several works have explored coarser equivalences and optimal algorithms for them.

In [Cha+17], the authors use a notion of observation of write operations and consider traces to be equivalent if the corresponding reads observe the same writes in both traces. However, their algorithm is optimal only for programs in which the graph of inter-thread communication is acyclic. A similar notion of observation is used in [AJLS18], in this case to only consider write events as interfering if at least one of them can be observed later. In [AAJN18] the authors use a notion of reads-from equivalence (i.e., trace equivalence based on program order and reads-from relation which connects reads to writes which produced the read value) for analysis of a fragment of the C11 memory model that contains only release and acquire memory orders. The advantage of this trace equivalence is that it does not distinguish traces which differ in the order of unobserved writes. The authors argue that working with this equivalence is easier for release-acquire ordering than for sequential consistency due to the complexity of checking if a reads-from relation can correspond to a run of a program. The same trace equivalence is used in [Abd+19] for sequentially-consistent programs. To avoid the need for expensive (NP-complete) checks for consistency of reads-from relations, the authors use two incomplete but polynomial algorithms. One of them can show that given relation is consistent and one which can show it is inconsistent, before running the expensive check which is exponential in the number of threads.

A somewhat different modification of DPOR is presented in [Alb+17]. It uses a notion of context-sensitive independence to improve on previous DPOR techniques. In essence, it considers two actions a and b independent in a given state if they can be executed in both as ab and ba and both executions lead to the same final state. This technique

[FG05] Flanagan et al.,
“Dynamic Partial-order
Reduction for Model
Checking Software”.

[SA07] Sen et al., “A
Race-Detection and Flip-
ping Algorithm for Auto-
mated Testing of Multi-
threaded Programs”.

[Tas+12] Tasharofi et al.,
“TransDPOR: A Novel
Dynamic Partial-Order
Reduction Technique for
Testing Actor Programs”.

[SKH12] Saarikivi et al.,
“Improving dynamic par-
tial order reductions
for concolic testing”.

[AAJS14] Abdulla et al.,
“Optimal Dynamic Par-
tial Order Reduction”.

[ZKW15] Zhang et al.,
“Dynamic Partial Or-
der Reduction for Re-
laxed Memory Models”.

[Cha+17] Chalupa et al.,
“Data-Centric Dynamic
Partial Order Reduction”.

[AAJN18] Abdulla et
al., “Optimal Stateless
Model Checking under the
Release-Acquire Semantics”.

[AJLS18] Aronis et
al., “Optimal Dynamic
Partial Order Reduc-
tion with Observers”.

[Abd+19] Abdulla et al.,
“Optimal Stateless Model
Checking for Reads-from
Equivalence under Se-
quential Consistency”.

[Maz87] Mazurkiewicz,
“Trace theory”.

[Alb+17] Albert et al.,
“Context-Sensitive Dynamic
Partial Order Reduction”.

uses a (local) state comparison and therefore is not entirely stateless. This approach was later combined with observers to achieve further reduction improvements [Alb+19].

Another reduction approach based on observation of read and written values is *Maximal Causality Reduction (MCR)* [Hua15], which can be seen as an alternative to DPOR. MCR employs an SMT solver to find new traces to explore, which allows it to explore fewer interleavings than Mazurkiewicz-trace-based DPOR techniques. Each time a new trace is found, it is guaranteed that at least one read will read a different value than read on already observed traces. An advantage of MCR is that it can be easily modified for parallel exploration (DPOR cannot be easily executed in parallel). The MCR was also applied to the TSO and PSO memory models [HH16].

Data Nondeterminism Most works concerning stateless model checking with DPOR expect that thread scheduling is the only source of nondeterminism in the system [FG05]. However, there are combinations of SMC with other techniques that can handle nondeterministic data. In [SKH12], a combination of DPOR with concolic execution is presented. Another combination of concolic execution with SMC is presented in [SA07], this one uses a different approach to DPOR than [FG05].

Tools and Techniques VeriSoft [God97; God05] is the pioneering tool of stateless model checking. It aims at safety verification of realistic parallel programs, primarily in C and C++, but also in other programming languages – VeriSoft works with compiled executable programs and uses a custom scheduler to explore (bounded) runs of these programs. To limit the re-exploration of states and state-space size, VeriSoft allows scheduling only on visible operations and uses partial order reduction.

jCUTE [SA07] is a tool that combines concolic execution and stateless model checking for Java programs, and therefore can handle both parallelism and data nondeterminism. To explore all possible behaviours of parallel programs, jCUTE detects data races (concurrent access of two threads to the same location, at least one of which is a write or a lock operation) and rearranges schedules which led to them.

CHESS [Mus+08] uses a custom scheduler to drive execution of compiled executable programs written mainly in C, C++ and .NET languages such as C#. It uses binary instrumentation to control programs' scheduling and record order of events. To limit state space explosion, it explores runs with fewer context switches first and uses a cache of happens-before graphs of events to avoid redundant explorations (and therefore is not entirely stateless). To handle the environment of the operating system better, CHESS can record and relay environmental values such as current time, process identifiers and output of the platform's random number generators. However, the user must ensure that other parts for the environment (e.g., file system, network) are used such that the program executes the same way if the same thread scheduling is repeated. The authors argue that this approach is practical for most unit tests.

[Alb+19] Albert et al., “Optimal Context-Sensitive Dynamic Partial Order Reduction with Observers”.

[Hua15] Huang, “Stateless Model Checking Concurrent Programs with Maximal Causality Reduction”.

[HH16] Huang et al., “Maximal Causality Reduction for TSO and PSO”.

[FG05] Flanagan et al., “Dynamic Partial-order Reduction for Model Checking Software”.

[SKH12] Saarikivi et al., “Improving dynamic partial order reductions for concolic testing”.

[SA07] Sen et al., “A Race-Detection and Flipping Algorithm for Automated Testing of Multi-threaded Programs”.

[God97] Godefroid, “Model Checking for Programming Languages Using VeriSoft”.

[God05] Godefroid, “Software model checking: The VeriSoft approach”.

[Mus+08] Musuvathi et al., “Finding and Reproducing Heisenbugs in Concurrent Programs.”

- [SKH12] Saarikivi et al., “Improving dynamic partial order reductions for concolic testing”.
- [KSH13] Kähkönen et al., “LCT: A Parallel Distributed Testing Tool for Multi-threaded Java Programs”.
- [DL15] Demsky et al., “SATCheck: SAT-Directed Stateless Model Checking for SC and TSO”.
- [ZKW15] Zhang et al., “Dynamic Partial Order Reduction for Relaxed Memory Models”.
- [Hua15] Huang, “Stateless Model Checking Concurrent Programs with Maximal Causality Reduction”.
- [HH16] Huang et al., “Maximal Causality Reduction for TSO and PSO”.
- [ND16] Norris et al., “A Practical Approach for Model Checking C/C++11 Code”.
- [CGS13] Christakis et al., “Systematic Testing for Detecting Concurrency Errors in Erlang Programs”.
- [AAJS14] Abdulla et al., “Optimal Dynamic Partial Order Reduction”.
- [AJLS18] Aronis et al., “Optimal Dynamic Partial Order Reduction with Observers”.
- [AAJL16] Abdulla et al., “Stateless Model Checking for POWER”.
- [Abd+17] Abdulla et al., “Stateless model checking for TSO and PSO”.
- [Abd+19] Abdulla et al., “Optimal Stateless Model Checking for Reads-from Equivalence under Sequential Consistency”.

LCT [SKH12; KSH13] is another tool which combines concolic execution and SMC to handle parallel Java programs with data non-determinism. It uses program instrumentation to add symbolic operations where needed and allows scheduling to happen only on visible actions. To speed up the analysis, LCT can run multiple executions of the program with different inputs in parallel and distribute them over the network.

SATCheck [DL15] is a stateless model checker for C programs with support for sequential consistency and TSO. It records dependencies between program events and uses SAT solver to reorder the events on a concrete run to get new interleavings with new behaviour.

rInspect [ZKW15] is an LLVM-based stateless model checker for C programs running under the TSO and PSO relaxed memory models. It uses store buffers and shadow threads that flush values from store buffers to the main memory. Store buffers can be optionally bounded. The shadow thread approach allows it to use existing DPOR techniques designed for sequential consistency.

The maximal causality reduction (MCR) is implemented in an unnamed tool for analysis of Java programs [Hua15; HH16]. It supports both sequential concurrency and the TSO and PSO relaxed memory models.

CDSChecker [ND16] is a stateless model checker for C and C++ with support for the C11/C++11 memory model (with the exception of release-consume synchronisation and out-of-thin-air values which are discouraged by the standard, but the C11/C++11 memory models allow them). It can simulate load speculation and delays by forwarding stored values to previous loads and validating this speculation. CDSChecker primarily aims to help with unit testing concurrent data structures.

Concuerror [CGS13; AAJS14; AJLS18] is an SMC for Erlang programs. In Erlang, the primary way of communication between processes is message passing. Concuerror uses context bounding and contains an implementation of the optimal DPOR for Mazurkiewicz traces and optimal DPOR based on observers.

Nidhugg [AAJL16; Abd+17; AJLS18; Abd+19], is a stateless model checker that focuses on C programs running under relaxed memory models. It has support for sequential consistency, TSO, PSO, POWER, and partially ARM memory models. For TSO and PSO, Nidhugg uses the optimal DPOR algorithm from [AAJS14] to explore exactly one execution from each equivalence class given by *chronological traces* which capture dependencies between memory operations. The combination of chronological traces and optimal DPOR algorithm means that for programs that do not exhibit relaxed behaviour under TSO/PSO, Nidhugg explores the same number of traces under TSO/PSO as it would explore under sequential consistency. For POWER, Nidhugg uses a different algorithm which can perform a redundant exploration of incomplete traces but does not generate redundant complete traces (for the trace equivalence given by Shasha-Snir traces [SS88]). Nidhugg also has an option to use a *reads-from*-based trace equivalence for sequential consistency, together with an optimal exploration algorithm for this equivalence. This equivalence does not distinguish traces that differ

only in the order of conflicting writes and therefore can be exponentially more coarse. The corresponding algorithm needs to decide if a given read-from relation is consistent, which is an NP-complete problem. However, the authors show that the expensive check can be avoided in most cases by use of faster but incomplete checks. Nidhugg uses LLVM to avoid the cost of direct analysis of C programs.

RCMC [KLSV17] is a stateless model checker for C and C++ programs running under the RC11 memory model [Lah+17] (which is an attempt to formalise and fix the C11 memory model). Instead of exploring interleavings, the tool uses execution graphs which represent visible program actions and dependencies between them.

Tracer [AAJN18] is a tool for analysis of the release-acquire fragment of the C11/C++11 relaxed memory model. It can work with C and C++ programs that do not use any other atomic orderings than release and acquire, i.e., it has no support for sequentially consistent or relaxed atomics. Tracer does not explore all possible orderings of conflicting writes; instead, the equivalence relation at the core of their DPOR implementation is given only by program orders and read-from to avoid redundancy. The proposed exploration algorithm is optimal for traces defined on this equivalence. In practice, this means that nondeterministic decisions take place at read operations and writes are only recorded, and backtracking is used for cases when a previous read might read from a later executed write.

[SS88] Shasha et al., “Efficient and Correct Execution of Parallel Programs That Share Memory”.

[KLSV17] Kokologiannakis et al., “Effective Stateless Model Checking for C/C++ Concurrency”.

[Lah+17] Lahav et al., “Repairing Sequential Consistency in C/C++11”.

[AAJN18] Abdulla et al., “Optimal Stateless Model Checking under the Release-Acquire Semantics”.

3.3 Symbolic and Concolic Execution

Symbolic execution was introduced in [Kin76]. It executes the program with symbolic values instead of concrete ones and tracks the relations between these symbolic values. When a symbolic value is used in a conditional branch, the symbolic executor can follow one or both of the branches based on the concrete values which are permitted by the symbolic values. If both of the branches are followed, the corresponding branching condition (or its negation) is added to the *path condition*. Therefore, the path condition collects the constraints required to get to the given code location. Symbolic executors usually use SAT or SMT solvers to decide which paths can be taken.⁵ Unlike previously discussed techniques, symbolic execution mostly targets sequential programs with inputs, i.e., it deals primarily with data nondeterminism and not scheduling nondeterminism. For this thesis, symbolic execution is notable because symbolic executors often have good support for real-world programs, including programs that use libraries and interact with their environment.

The state space explored by a symbolic executor forms a tree with branches at conditions. This *symbolic execution tree* can be quite large even for small sequential programs with loops or recursion – this is the problem known as *path explosion*.⁶ The symbolic execution tree can be explored in different manners (e.g., depth-first search, random search) depending on the objective of the analysis, for example, if it is desirable to explore all the behaviours of the program, or if some representative sample should be analysed for the purpose of testing.

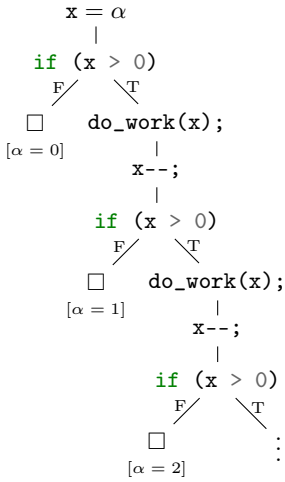
[Kin76] King, “Symbolic Execution and Program Testing”.

⁵ SAT solvers are used for propositional formulas while SMT solvers can solve first-order formulas that use some theory, for example, the theory of bitvectors which can be used to precisely model computer integers. The advantage of SMT solvers is that the encoding is simpler and the solver can make use of the additional information encoded in the richer theory.

⁶ Consider this code:

```
unsigned x = input();
while (x > 0) {
    do_work(x);
    x--;
}
```

Its symbolic execution tree has as many branches as there are values in the **unsigned** data type:



[CDE08] Cadar et al., “KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs”.

[GKS05] Godefroid et al., “DART: Directed Automated Random Testing”.

[SMA05] Sen et al., “CUTE: A Concolic Unit Testing Engine for C”.

[MS07] Majumdar et al., “Hybrid Concolic Testing”.

Test Generation Symbolic execution can be used to generate concrete test cases for a program (or a function) – each time the executor finishes a path (either the program terminates, or an error is found), it can use a solver to generate concrete inputs that reproduce this path in a test case [CDE08]. This way, it is possible to generate tests which cover the program well and do not execute the same path repeatedly. Even if the symbolic execution tree is large or infinite, the test generation can use random search or some other heuristics to provide good coverage of the code and therefore improve on manually-written tests. Furthermore, tests generated from symbolic execution can be very useful for bug fixing as the programmer can use their debugging tools of choice on the generated test and do not need to learn a verifier-specific way to analyse the counterexample.

Concolic Execution *Concolic Execution*, also known as *Concolic Testing*, *Dynamic Symbolic Execution* or *Directed Automated Random Testing* is a modification of symbolic execution which can eliminate some of the expensive solver queries issued during exploration [GKS05; SMA05]. With concolic execution, the program is executed with concrete values and it follows a single run, but the concolic executor records the path condition as with conventional symbolic execution. When the run finishes, the executor uses the path condition and a solver to find values of input variables that will lead the program to follow a different path. This is repeated until no more paths can be discovered, or until the given coverage criteria are met.

Tools CUTE [SMA05; MS07] is a concolic test generator for sequential C programs that specializes at dynamic memory. It can generate input values for pointers (e.g., linked list nodes) as well as scalars. CUTE uses approximate constraints for pointers to prevent undecidable theories (pointer constraints can contain pointer (in)equality and check for NULL pointers). It detects assertion violations, memory access errors, and certain cases of infinite loops. Furthermore, CUTE has support for *hybrid concolic testing*, which combines concolic and random testing. It targets reactive programs and primarily uses random-generated inputs. In this mode, concolic input generation is used to avoid getting stuck (i.e., the tool will switch to concolic input generation if it fails to add to coverage in a certain number of randomized tests).

KLEE [CDE08] is a symbolic executor and test generator that targets realistic C and C++ programs that communicate with the environment. Its goal is to work with unmodified programs (with source code available, the program needs to be compiled to LLVM). KLEE can check for memory safety and assertion safety violations. It uses array theory with a separate array for each memory object to facilitate analysis of programs with dynamic memory. In KLEE, symbolic pointers are handled by explicit enumeration of all possibilities when the pointer is dereferenced. To achieve good coverage even when time and the size of the symbolic execution tree do not allow for full exploration, KLEE uses path selection heuristics. KLEE is notable for its support of realistic programs that communicate with their environment. It has models of

some of the POSIX system calls (mostly related to the file system) and these models can contain symbolic values (e.g., in files). The system call models are written in C and can be modified without modification of KLEE itself. Furthermore, the program running in KLEE can also access the real file system, and KLEE can perform fault injection into system calls. When an error is encountered, KLEE generates a test harness which creates the corresponding files and fills them with values which reproduce the problem. To support the C standard library, KLEE uses uClibc standard library implementation, which is linked to the program under analysis.

Symbiotic [CSV18; Cha+20] is a tool for analysis of sequential C programs which uses KLEE symbolic executor as a backend. It can detect assertion violations, memory safety errors, memory leaks, and integral overflows. Symbiotic uses instrumentation to add pointer checks where needed and uses various analysis techniques (e.g., pointer analysis and shape analysis based on Predator [DPV13]) to limit the number of inserted checks. Furthermore, Symbiotic uses program slicing to remove code irrelevant for the property. Symbiotic has basic support for the detection of termination and nontermination. It can detect simple cycles in the program’s state space and therefore in some cases decide the program does not terminate (for non-nested loops that preserve values of all variables). It can also conclude the program terminates if all paths were explored.

Pinaka [CJ19] is a symbolic executor for sequential C programs which uses incremental solver to check path feasibility. It can run either in a fully incremental mode where it reuses one solver instance for the whole program, which can cause big formulas or in a partially incremental mode which creates a new instance of the solver on backtracking. Pinaka may not terminate on nonterminating programs, and it has an option for loop unwinding limit. It can be used to show termination – if a program is found to be safe without any unwinding limit, then all its paths terminate.

JDart [MH20; Luc+16] is a concolic executor and test generator for sequential Java programs build on Java Pathfinder. It can detect assertion violations and uncaught exceptions. JDart attempts to find small values for symbolic variables, which can make loops depending on symbolic values shorter.

Java Ranger [Sha+20] is a symbolic executor for Java based on Symbolic Pathfinder. It can merge paths using summarization of regions with multiple paths (e.g., branches of an `if` statement).

COASTAL [VG20] combines concolic execution with fuzz testing for Java programs. It uses fuzz testing for fast exploration; the fuzzing mode only tracks the direction of branches. Concolic execution can be used to get to parts of the program’s state space which are harder to reach. COASTAL can explore different parts of the symbolic execution tree in parallel (with different fuzzer or concolic executor instances). To help with the analysis of realistic programs, COASTAL has models for some of the standard Java classes.

Map2Check [RMCB20] is a tool which combines fuzzing, symbolic execution, inductive invariants for safety checking for sequential C

[CSV18] Chalupa et al., “Joint Forces for Memory Safety Checking”.

[Cha+20] Chalupa et al., “Symbiotic 7: Integration of Predator and More”.

[DPV13] Dudka et al., “Byte-Precise Verification of Low-Level List Manipulation”.

[CJ19] Chaudhary et al., “Pinaka: Symbolic Execution Meets Incremental Solving”.

[MH20] Mues et al., “JDart: Dynamic Symbolic Execution for Java Bytecode (Competition Contribution)”.

[Luc+16] Luckow et al., “JDart: A Dynamic Symbolic Analysis Framework”.

[Sha+20] Sharma et al., “Java Ranger at SV-COMP 2020 (Competition Contribution)”.

[VG20] Visser et al., “COASTAL: Combining Concolic and Fuzzing for Java (Competition Contribution)”.

[RMCB20] Rocha et al., “Map2Check: Using Symbolic Execution and Fuzzing”.

[BCCZ99] Biere et al.,
“Symbolic Model Check-
ing without BDDs”.

⁷ For tools which use ap-
proximation or heuristics,
checking for termination
is not the same as check-
ing for nontermination
as the tool might simply
return *unknown* as a re-
sult if it fails to prove its
objective (i.e., termina-
tion or nontermination).

⁸ Static single assign-
ment; each variable is
assigned only at one
place in the source code.

[CKL04] Clarke et al.,
“A Tool for Checking
ANSI-C Programs”.

[CF11] Cordeiro et
al., “Verifying Multi-
Threaded Software Us-
ing SMT-based Context-
Bounded Model Checking”.

[RG05] Rabinovitz et al.,
“Bounded Model Checking
of Concurrent Programs”.

[AKT13] Alglave et al.,
“Partial Orders for Efficient
Bounded Model Checking
of Concurrent Software”.

[GG08] Ganai et al., “Ef-
ficient Modeling of Con-
current Systems in BMC”.

[QW04] Qadeer et al.,
“KISS: Keep It Sim-
ple and Sequential”.

[LR09] Lal et al., “Reduc-
ing concurrent analysis
under a context bound
to sequential analysis”.

[Inv+15] Inverso et al.,
“Lazy-CSeq: A Context-
Bounded Model Check-
ing Tool for Multi-
threaded C-Programs”.

[Tom+16] Tomasco et
al., “MU-CSeq 0.4: In-
dividual Memory Lo-
cation Unwindings”.

programs. It is LLVM based and uses Klee as a backend symbolic executor. The idea of this combination is that fuzzing is used to find shallow bugs, while symbolic execution finds deep bugs. Map2Check runs multiple fuzzers in parallel.

3.4 Bounded Model Checking & Other Symbolic Techniques

Bounded model checking (BMC) was introduced in [BCCZ99] as an alternative to symbolic model checking techniques which used binary decision diagrams. BMC encodes program runs of fixed length into propositional formulas, or to formulas over some first-order logic. It then relies on SAT or SMT solvers to decide whether the formula is satisfiable or not. One of the advantages of BMC is that it naturally handles data nondeterminism and can cope with scheduling nondeterminism too.

The original paper introducing BNC presented a verification procedure for LTL (which detected loops in the program runs by checking equality of the last state of a bounded run to some of its previous states). However, for realistic programs, the focus is mainly on safety properties, and more recently also on (non)termination.⁷

BMC tools usually build formula from an SSA form⁸ of the program with a bounded number n of loop iterations and recursion depth [CKL04]. The loops and recursive functions are unwound – repeated n times with guard before each repetition which allows the program to skip the repetition if the number of iterations was lower than n . Furthermore, *unwinding assertions* are inserted after the last repetition to ensure the unwinding was sufficient. The result is that all jump lead forward, which simplifies the creation of the logic formula which describes the transition function of the program.

Parallelism We have categorized the existing approaches to parallelism in BMC into three broad categories. The conceptually simplest is to explicitly enumerate symbolic context-switch bounded runs of the program and then encode them to resolve symbolic data. This approach is used by ESBMC [CF11] and by [RG05] for an early concurrent extension of CBMC. Another approach is to encode the control flow of threads separately and add scheduling constraints which specify inter-thread ordering. This approach is used by CBMC [AKT13] and also proposed in [GG08]. The last approach is sequentialization, which makes use of a BMC for sequential programs and a preprocessing step that converts a parallel program into a nondeterministic sequential program. Many sequentialization schemes were proposed, differing both in the size of the encoding and the way they limit thread interaction (e.g., limit to the number of context switches, to the number of shared-variable interactions) [QW04; LR09; Inv+15; Tom+16; Tom+17].

Path Bounding The main limitation of BMC is that it explores only runs up to some length bound k . Therefore, BMC alone cannot prove correctness unless it shows that the loop bound was sufficient (for example, by showing unwinding assertions are not violated). To

mitigate this limitation, BMC can be combined with other techniques which aim to prove correctness of the program, such as k -induction or IC3.

k -Induction k -induction [DHKR11; GIC17] is an extension of bounded model checking which allows it to find bugs faster and to prove correctness of programs without unwinding loops fully. Multiple k -inductions schemes exist, in the one proposed in [GIC17] a verifier with k -induction repeatedly performs following three steps with an increasing bound k .

- *Base Case* checks if an error is reachable in k steps.
- *Forward Condition* checks if all program's paths had terminated in k steps (i.e., the tool had already explored all states, and therefore the verification is done).
- *Inductive Step* checks a formula corresponding to the following proposition: if the property holds for the first k steps, it also holds for $k+1$ steps. This step is checked in a modified program in which all loop variables are initialised to nondeterministic values before the loops, and the loops are required to execute to completion.⁹

One of the problems with k -induction is that the induction argument may not be strong enough to prove the inductive step. For this reason, the induction can be strengthened by invariants derived from the original program (i.e., the invariant constraints the nondeterministic values of loop variables) [RICB17].

IC3/PDR Another symbolic technique that uses solvers is *IC3* (*Incremental Construction of Inductive Clauses for Indubitable Correctness*), also known as *Property Directed Reachability (PDR)*. It verifies a system without unrolling its transition relation and uses a lot of relatively small solver queries (while BMC usually encodes the whole program into one big query). IC3 iteratively builds approximations of reachable safe state space and refines them once it finds a step that leads from the set of already discovered states to an unsafe state. The refinement works backwards from the predecessor of the unsafe state. If the refinement reaches the initial state, the analysed system is incorrect. Solver queries in IC3 concern separate steps in the state space (not entire paths).

IC3 was initially introduced in the context of hardware (circuit) verification [Bra11] and refined later as *property directed reachability (PDR)* [EMB11]. It was later extended to software using SMT solvers [CG12]. The SMT-based IC3 can also handle software with infinite state space. Software IC3 was also combined with predicate abstraction in multiple ways including *Implicit Predicate Abstraction* [CGMT14] and *Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR)* [BBW14]. IC3 was also combined with k -induction to strengthen the inductive step of k -induction [BD20].

[Tom+17] Tomasco et al., "Using Shared Memory Abstractions to Design Eager Sequentializations for Weak Memory Models".

[DHKR11] Donaldson et al., "Software Verification Using k -Induction".

[GIC17] Gadelha et al., "Handling loops in bounded model checking of C programs via k -induction".

⁹ This can be achieved by inserting `assume` statements that ensure that the loop body is entered and that after the given number of iterations the condition for loop termination holds.

[RICB17] Rocha et al., "Model Checking Embedded C Software Using k -Induction and Invariants".

[Bra11] Bradley, "SAT-Based Model Checking without Unrolling".

[EMB11] Een et al., "Efficient Implementation of Property Directed Reachability".

[CG12] Cimatti et al., "Software Model Checking via IC3".

[CGMT14] Cimatti et al., "IC3 Modulo Theories via Implicit Predicate Abstraction".

[BBW14] Birgmeier et al., "Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR)".

[BD20] Beyer et al., "Software Verification with PDR: An Implementation of the State of the Art".

[BCCZ99] Biere et al., "Symbolic Model Checking without BDDs".

[Iva+05] Ivančić et al.,
“F-Soft: Software Ver-
ification Platform”.

¹⁰ BDDs, or Binary De-
cision Diagrams are used
in symbolic model check-
ing methods. However,
for software analysis these
methods are superseded by
bounded model checking
and not used these days.

[CKL04] Clarke et al.,
“A Tool for Checking
ANSI-C Programs”.

[KT14] Kroening
et al., “CBMC – C
Bounded Model Checker”.

[AKT13] Alglave et al.,
“Partial Orders for Efficient
Bounded Model Checking
of Concurrent Software”.

[Gad+18] Gadelha et al.,
“ESBMC 5.0: An Industrial-
Strength C Model Checker”.

[GMCN19] Gadelha et
al., “ESBMC v6.0: Ver-
ifying C Programs Us-
ing k-Induction and
Invariant Inference”.

[CF11] Cordeiro et
al., “Verifying Multi-
Threaded Software Us-
ing SMT-based Context-
Bounded Model Checking”.

[Ram+13] Ramalho
et al., “SMT-Based
Bounded Model Check-
ing of C++ Programs”.

[SCL15] Sousa et al.,
“Bounded model checking
of C++ programs based
on the Qt framework”.

[GMCL16] Garcia et al.,
“ESBMC^{QtOM}: A Bounded
Model Checking Tool to
Verify Qt Applications”.

Tools The original bounded model checker BMC [BCCZ99] was an LTL bounded model checker that targeted a modelling language.

F-Soft [Iva+05] is a tool for safety analysis of sequential C programs. It supports both BMC and BDD-based model checking.¹⁰ It uses program slicing and predicate abstraction to improve analysis performance. An interesting property of F-Soft is that it can generate executable counterexamples.

CBMC [CKL04; KT14] is a widely used SAT-based bounded model checker for sequential and parallel C programs. It uses bit-precise encoding and therefore preserves the semantics of computer integers. CBMC can detect safety errors, and it uses instrumentation for detection of errors other than assertions. For efficient analysis of parallel programs, CBMC uses encoding based on partial orders described in [AKT13] – it builds the formula from the SSA form of each thread and ordering constraints for shared variables. CBMC is notable for its support of relaxed memory models, including x86-TSO, PSO and partially the POWER memory model, as well as sequential consistency [AKT13]. CBMC uses a custom parser for C.

ESBMC [Gad+18; GMCN19], is another widely used BMC for C programs. It can detect assertion violations, memory errors, overflows and mutex-caused deadlocks. ESBMC was derived from CBMC, but it uses SMT for the encoding of the program and has support for k -induction with invariant generation (based on interval analysis of integral variables). The invariant generation is currently unsound for programs with pointers. The support for concurrency in ESBMC is described in [CF11]. ESBMC explores possible program interleavings explicitly and collects constraints on nondeterministic variables, which are then used to generate a formula. The formula can be generated and solved either for each run separately (lazy approach, has the advantage of being incremental), or all interleavings can be encoded into one formula that uses guards to encode scheduling of each interleaving (schedule recording). There is also an encoding based on under-approximation and widening that attempts to produce more compact encoding. The approximation uses proofs of unsatisfiability of the formula for refinement. According to [CF11], the lazy approach seems to work best. Regardless of the formula encoding, ESBMC uses a bound on the number of context switches. To limit the number of runs which need to be explored, ESBMC allows context switches only at visible instructions (i.e., instructions which access shared memory). The handling of parallelism in ESBMC makes it closely related to explicit-state model checkers with symbolic extensions, like Symbolic Pathfinder or DIVINE. Similarly to CBMC, ESBMC used a custom C parser, but it had later switched to using the clang compiler for parsing C, and there is ongoing work for using this parser for C++. Unlike other tools which use clang, ESBMC does not use the LLVM intermediate representation but instead builds its internal representation based on the C/C++ AST produced by clang. Furthermore, ESBMC has derivatives which aim at C++ verification. ESBMC++ [Ram+13] uses a model of parts of the standard C++ library, has support for C++ exceptions and inheritance. ESBMC^{QtOM} [SCL15; GMCL16] adds a model of parts of the Qt framework for C++

programs. In both cases, these models approximate behaviour of the respective libraries based on the standard or documentation.

JBMC [CKS19; Cor+18], is a bounded model checker for Java built on the same basis as CBMC. It uses a combination of SAT and SMT solving with a dedicated solver for string operations. JBMC has support for exceptions (by lowering them to jumps in its intermediate representation) and has an exact verification-friendly model of most common parts of the Java standard libraries. It can detect assertions, memory errors, integral overflows and type casting errors. JBMC has currently no support for threads, and it also lacks support for the Java Native Interface (JNI) and reflection.

LLBMC [MFS12; FMS13] is an SMT-based, bit-precise BMC for sequential C and C++ programs that is notable for its use of the LLVM intermediate representation which allows it to reuse existing C and C++ compilers. It is also one of the few tools which claim support of C++. However, the C++ support has some limitations, most notably, there is no support for exceptions and run-time type support. Furthermore, LLBMC has no support for threads and floating-point arithmetic in both C and C++. It can detect assertion violations, integer overflows, invalid shifts, memory errors and memory leaks.

Another LLVM-based tool is LLVMVF [SS13], a generic verification framework on which a BMC for parallel programs is built. Concurrency support in LLVMVF is incomplete; for example, it lacks support for condition variables.

An older version of Map2Check [RBC15] was a test generator for memory safety and leak detection unit tests for C programs. It uses ESBMC to extract verification conditions (conditions of memory safety violation). Then it instruments the program to convert memory safety to assertions about pointers and runs the tests with concrete data.

Yogar-CBMC [Yin+18; YDLW19] is a derivative of CBMC that uses abstraction and refinement for the encoding of scheduling constraints in parallel programs to make the formulas smaller. Unlike CBMC, it has no support for relaxed memory. It can run multiple counterexample-guided refinement loops in parallel, and they share the learned scheduling constraints to analyse the program faster. Yogar-CBMC was able to solve all concurrency benchmarks in SV-COMP 2019 and significantly outperformed CBMC.

CBMC-Path [KT19] is another derivative of CBMC. It encodes paths in the state space one by one instead of building a formula for the whole (bounded) state space of a parallel program. While it was designed to facilitate faster bug discovery for SV-COMP, it performed significantly worse than CBMC.

DepthK [RICB17; Roc+17], is a tool based on ESBMC with the addition of polyhedral invariants which aim to strengthen inductive step in the k -induction. It was presented before ESBMC gained the ability to compute invariants itself, and it uses an external invariant generator. DepthK can analyse concurrent C programs.

Dartagnan [PFHM20; Gav+19] is a BMC for analysis of parallel C programs under relaxed memory models. In this case, the relaxed memory model is specified on input, not encoded in the verifier. To

[CKS19] Cordeiro et al., “JBMC: Bounded Model Checking for Java Bytecode”.

[Cor+18] Cordeiro et al., “JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode”.

[MFS12] Merz et al., “LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR”.

[FMS13] Falke et al., “The bounded model checker LLBMC”.

[SS13] Sousa et al., “LLVMVF: A generic approach for verification of multicore software”.

[RBC15] Rocha et al., “Memory Management Test-Case Generation of C Programs Using Bounded Model Checking”.

[Yin+18] Yin et al., “YOGAR-CBMC: CBMC with Scheduling Constraint Based Abstraction Refinement”.

[YDLW19] Yin et al., “Parallel Refinement for Multi-Threaded Program Verification”.

[KT19] Khazem et al., “CBMC Path: A Symbolic Execution Retrofit of the C Bounded Model Checker”.

[RICB17] Rocha et al., “Model Checking Embedded C Software Using k-Induction and Invariants”.

[Roc+17] Rocha et al., “DepthK: A k-Induction Verifier Based on Invariant Inference for C Programs”.

[PFHM20] Ponce-de-León et al., “Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution)”.

[Gav+19] Gavrilenko et al., “BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings”.

[RE14] Rakamarić et al., “SMACK: Decoupling Source Language Details from Verifier Implementations”.

[Inv+15] Inverso et al., “Lazy-CSeq: A Context-Bounded Model Checking Tool for Multithreaded C-Programs”.

[Ngu+17] Nguyen et al., “Lazy-CSeq 2.0: Combining Lazy Sequentialization with Abstract Interpretation”.

[IT20] Inverso et al., “Parallel and Distributed Bounded Model Checking of Multithreaded Programs”.

[CCM09] Canet et al., “A Value Analysis for C Programs”.

[LR09] Lal et al., “Reducing concurrent analysis under a context bound to sequential analysis”.

[Tom+15] Tomasco et al., “Verifying Concurrent Programs by Memory Unwinding”.

[Tom+16] Tomasco et al., “MU-CSeq 0.4: Individual Memory Location Unwindings”.

[Tom+17] Tomasco et al., “Using Shared Memory Abstractions to Design Eager Sequentializations for Weak Memory Models”.

[CDS13] Cho et al., “Blitz: Compositional bounded model checking for real-world programs”.

[RG05] Rabinovitz et al., “Bounded Model Checking of Concurrent Programs”.

achieve this, Dartagnan uses symbolic encoding of relations between the program’s actions (data dependencies, reads-from, program order, ...) and the memory model restricts the possible runs based on these relations. It should be possible to use Dartagnan with a wide range of memory models including x86-TSO, ARMv8, POWER, C/C++11 and Linux kernel memory model. To speed up the analysis, it uses relation analysis that reduces the size of the encoding – it determines which pairs of events may influence constraints specified by the memory model. Dartagnan internally uses Boogie intermediate language and translates C to it using SMACK [RE14]; it lacks support for pointer arithmetic.

Lazy-CSeq [Inv+15; Ngu+17; IT20] is a sequentialisation-based tool for analysis of concurrent C programs which uses bounded model checker as a backend (CBMC by default). It performs source-to-source transformation from a concurrent C program to a sequential C program. The sequential program simulates a bounded number of round-robin scheduling rounds (i.e., the number of context switches is limited, and the threads are required to execute in a fixed order, with the possibility to perform no steps in a given round). This way, Lazy-CSeq can simulate a bounded number of interleavings with small memory overhead while limiting added nondeterminism. It can detect errors detectable by the used backend and POSIX mutex deadlocks. To speed up analysis, Lazy-CSeq uses Frama-C [CCM09] to infer bounds of integral variables and shrink the corresponding bitvector formulas. Lazy-CSeq performs *lazy sequentialisation*, i.e., it explores only the reachable state space of the program, unlike some older works on sequentialisation like [LR09]. Lately, support for parallelisation by partitioning the set of execution traces into independent instances was added to Lazy-CSeq.

MU-CSeq [Tom+15; Tom+16] and IMU-CSeq [Tom+17] are sequentialisation-based tools for analysis of concurrent C programs that use CBMC as a backend. They work by nondeterministically guessing a bounded sequence of shared memory writes and then simulating the program so that its runs match this sequence. Optionally, they can also execute unobserved memory writes which are not part of the sequence. The difference between the tools is that IMU-CSeq uses a separate sequence for each memory location, while MU-CSeq has a single sequence of visible memory operations. IMU-CSeq also has support for relaxed memory (TSO, PSO) using *Shared Memory Abstraction*, which defines the behaviour of memory operations and synchronisation primitives of the given memory model. These tools perform *eager sequentialisation*, i.e., threads are executed separately, with nondeterministic values for shared memory reads which connect them.

BLITZ [CDS13] is a BMC for C programs that decomposes the verification instance using approximations of preconditions of property violation. These approximations are gradually refined based on the proofs of unsatisfiability for the under-approximated instances. It aims to detect assertion violations and memory safety in larger programs (100kloc). BLITZ appears to have no support for parallelism.

TCBMC [RG05] is an early extension of CBMC to parallel programs.

It uses context switch bounding, and the authors proposed mutex deadlock detection and race detection for it. However, the implementation supported only two threads without deadlock detection.

CheckFence [BAM07] is a tool for checking implementations of concurrent data structures in C against a relaxed memory model which is an over-approximation of many hardware memory models (i.e., safety in CheckFence should imply safety under these memory models). It uses a test program as a specification – it checks if the set of all executions under the given memory model is a subset of a set of *serial executions* which are sequentially consistent executions in which each operation with the data structure is atomic. Therefore, CheckFence does not detect properties such as assertion violation or memory safety but considers the program unsafe if it can exhibit new behaviour under its relaxed memory model. CheckFence encodes threads and scheduling separately.

NBIS [GW14] is an incremental BMC for sequential C programs. In this case, incremental means that it can change bounds without throwing away the formula and re-running the solver (provided the solver can support incremental solving, which is common). NBIS uses LLVM IR to facilitate analysis of C programs.

VVT [GLW16; GLSW17] is a successor to NBIS which combines bounded model checking for fast bug discovery with CTIGAR, an SMT-based version of IC3 with counterexample refinement. It aims at the analysis of parallel C software and uses LLVM IR internally. VVT can analyse infinite-state systems (with a finite number of threads and with arrays with statically known bounds). To improve performance with parallel programs, VVT uses dynamic state-space reduction based on conditionally atomic blocks of code (which are instrumented into the code). VVT is not bit-precise, it uses integer arithmetics instead of arithmetics with overflows and therefore cannot find problems caused by bounded bitwidth of computer integers.

power2sc [AABN17] is a tool for analysis of C programs under the POWER memory model which uses CBMC as a backend. It uses program transformation to build a new program which, when executed under sequential consistency, simulates context-switch-bounded runs of the original program under POWER. The transformed program relies on nondeterminism to guess the results of interactions between threads and then checks the validity of these guesses. While the presented tool uses CBMC, the authors claim that their program transformation can work with any verifier for safety-checking parallel program under sequential consistency.

[BAM07] Burckhardt et al., “CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models”.

[GW14] Günther et al., “Incremental Bounded Software Model Checking”.

[GLW16] Günther et al., “Vienna Verification Tool: IC3 for Parallel Software”.

[GLSW17] Günther et al., “Dynamic Reductions for Model Checking Concurrent Software”.

[AABN17] Abdulla et al., “Context-Bounded Analysis for POWER”.

3.5 Other Approaches to Program Analysis

There are many program analysis techniques relevant to realistic parallel programs which were not covered in the previous sections. These techniques are often very different, and categorising them does not seem to provide many benefits. Therefore, we will now outline these techniques as they are represented by notable tools that implement them.

[Hei+17] Heizmann et al., “Ultimate Automizer with an On-Demand Construction of Floyd-Hoare Automata”.

[HHP13] Heizmann et al., “Software Model Checking for People Who Love Automata”.

[Die+20] Dietsch et al., “Ultimate Taipan with Symbolic Interpretation and Fluid Abstractions”.

[Gre+17] Greitschus et al., “Ultimate Taipan: Trace Abstraction and Abstract Interpretation”.

[NDMP15] Nutz et al., “ULTIMATE KOJAK with Memory Safety Checks”.

[Cas+17] Cassez et al., “Skink: Static Analysis of Programs in LLVM Intermediate Representation”.

[AAJS14] Abdulla et al., “Optimal Dynamic Partial Order Reduction”.

[DLW15] Dangl et al., “CPAchecker with Support for Recursive Programs and Floating-Point Arithmetic”.

[BK11] Beyer et al., “CPAchecker: A Tool for Configurable Software Verification”.

[BD20] Beyer et al., “Software Verification with PDR: An Implementation of the State of the Art”.

[And+17] Andrianov et al., “CPA-BAM-BnB: Block-Abstraction Memoization and Region-Based Memory Models for Predicate Abstractions”.

Tools Ultimate Automizer [Hei+17; HHP13] is a program analysis tool for sequential and parallel C programs. It uses *automata-theoretic verification approach* which, in essence, converts a program to an automaton (starting with an automaton derived from the control-flow graph), and then checks reachability of error states in this automaton and possibly refines it. The automata are constructed in such a way that the set of traces of the automaton over-approximates program traces. Therefore, if the automaton cannot reach an error location, the program is correct. If the automaton can reach an error location, the feasibility of the error trace is checked, and if it is not feasible, the automaton is refined (otherwise an error was found). Therefore, automata-theoretic verification is an instance of CEGAR. The refinement is performed by the construction of automaton of spurious traces – Floyd-Hoare automaton – which is constructed on-demand using an SMT solver. Furthermore, Ultimate Automizer uses nested words automata for interprocedural analysis. It also uses Büchi automata and ranking functions for termination and nontermination analysis. Ultimate Automizer can detect assertion violations, memory safety errors, check for integral overflows, and termination.

Ultimate Taipan [Die+20; Gre+17] is a close relative of Ultimate Automizer that uses refinement based on *path programs*. When a possible error is found, it projects the original program into the error trace to obtain a corresponding path program that can contain loops and only contains statements found on the trace. It then attempts to show trace infeasibility by proving unreachability of the error location in the path program using invariants.

Ultimate Kojak [NDMP15] is another related tool which uses a different algorithm for the refinement step. This algorithm is based on proofs of unsatisfiability provided by an SMT solver used to check the feasibility of error paths.

Skink [Cas+17] is an LLVM-based tool for assertion checking of sequential and concurrent C programs. It uses the automata-theoretic verification approach similar to Ultimate Automizer (with automata-based refinement). However, Skink is limited to programs which can be fully inlined, i.e., it does not support function calls. Furthermore, it uses mathematical integers and therefore is not bit-precise. To improve efficiency with parallel programs, Skink uses reduction based on the source-DPOR algorithm used more often in stateless model checking [AAJS14].

CPAchecker [DLW15; BK11; BD20] is a program analysis tool for C programs and a modular framework which aims at easy integration of new components and research of new verification ideas. It uses a combination of abstract domains and has support for various techniques such as k -induction, predicate abstraction, and property directed reachability (PDR/IC3). To avoid false counterexamples, CPAchecker performs bit-precise validation of counterexamples. Various tools build on the CPAchecker framework are participating in the SV-COMP. It appears that CPAchecker without additional extensions participates in SV-COMP under the name CPA-Seq.

CPA-BAM-BnB [And+17] is a CPAchecker-based tool that uses

Block Abstraction Memoization, which means it divides programs into blocks (often a block is a function) and analyses blocks separately. It uses a cache of existing block abstractions. Furthermore, it uses value analysis and predicate analysis with refinement on spurious counterexamples. It also uses BnB, which is a model of memory that uses predicate analysis and uninterpreted functions to map memory locations to memory values. In this model, memory is split into regions, and each region has its mapping function.

CPALockator [AMK18] is a version of CPAchecker which specialises on parallel programs and data race detection. CPALockator aims to provide a lightweight analysis at the cost of possibly missing errors. To do this, it tracks locks and threads that are active at the given time. CPALockator uses predicate analysis and CEGAR to exclude spurious race conditions.

PeSCo [RW19; CHJW17] is a tool build on top of CPAchecker that tries to predict efficient verification approach for a given task using machine learning. The learning uses graph encoding of programs and decides which of the six predefined configurations of CPAchecker should run and in which order.

Corral [LQL12] is a verifier for the Boogie modelling language accompanied by language frontends for C and .NET bytecode. It performs bounded reachability and uses abstraction and refinement. Abstracted programs track values of only a selected set of variables (which is initially empty). On top of that, functions are inlined on-demand if an over-approximation of a function's effects causes a spurious error. Corral supports parallel programs using sequentialisation.

SMACK [RE14; Car+16; GBHR20] is a bug-finder for various programming languages that can be translated to LLVM. It translates LLVM IR into Boogie intermediate verification language and then uses Boogie, Corral (default) or other Boogie-based verifiers as a backend. This approach is intended to allow easy prototyping of backend verifiers which do not need to handle some of the complexities of programming languages. SMACK can use either bitvectors (i.e., be bit-precise) or arbitrary precision integers. Since Boogie has no support for dynamic memory, SMACK partitions memory into disjoint regions that are represented by maps in Boogie. While SMACK initially targeted C, it was later extended to (partially) support various other programming languages that can be translated to LLVM: C++, Rust, Objective-C, D, Fortran, Swift, and Kotlin. In this effort, SMACK relies heavily on LLVM-based language frontends to provide basic language support. On top of that, SMACK has basic support for standard libraries (allocation, C string and math operations, POSIX threads, models of most common parts of the libraries). However, the authors claim these models must be written manually, which they consider being a major undertaking. This is shown to be a limiting factor, especially for languages with heavy runtime components, such as Swift, Objective-C and Kotlin. Furthermore, thanks to compilation to a common intermediate representation (LLVM), SMACK has support for cross-language program analysis. With a concurrency-enabled backend (e.g., Corral), Smack also supports parallel programs.

[AMK18] Andrianov et al., “Predicate Abstraction Based Configurable Method for Data Race Detection in Linux Kernel”.

[RW19] Richter et al., “PeSCo: Predicting Sequential Combinations of Verifiers”.

[CHJW17] Czech et al., “Predicting Rankings of Software Verification Tools”.

[LQL12] Lal et al., “A Solver for Reachability Modulo Theories”.

[RE14] Rakamarić et al., “SMACK: Decoupling Source Language Details from Verifier Implementations”.

[Car+16] Carter et al., “SMACK Software Verification Toolchain”.

[GBHR20] Garzella et al., “Leveraging Compiler Intermediate Representation for Multi- and Cross-Language Verification”.

[KRS19] Kahsai et al., “JayHorn: A Java Model Checker”.

[GPR11] Gupta et al., “Threader: A Constraint-Based Verifier for Multithreaded Programs”.

[Jon83] Jones, “Tentative Steps toward a Development Method for Interfering Programs”.

[OG76] Owicki et al., “An axiomatic proof technique for parallel programs I”.

[PŠV20] Peringer et al., “PredatorHP Revamped (Not Only) for Interval-Sized Memory Regions and Memory Reallocation (Competition Contribution)”.

[DPV13] Dudka et al., “Byte-Precise Verification of Low-Level List Manipulation”.

[CKSY05] Clarke et al., “SATABS: SAT-Based Predicate Abstraction for ANSI-C”.

¹¹ Bit-blasting is a technique which rewrites bitvector variables to a sequence of boolean variables and implements bitvector operations by respective boolean circuits. It allows solving (existentially-qualified) bitvector formulas with a SAT solver.

[AKNT13] Alglave et al., “Software Verification for Weak Memory via Program Transformation”.

JayHorn [KRS19] is a tool for analysis of sequential Java programs. It encodes programs into Horn clauses and can handle some instances of unbounded heap data structures and unbounded execution using invariants. JayHorn is not bit precise (it uses unbounded integers) and has only a basic model of Java library (it mostly assumes library calls return arbitrary values).

Threader [GPR11] is a tool for compositional verification of parallel C programs. It uses thread-modular proofs based on proof rules from [Jon83; OG76]. The analysis is based on abstract reachability with refinement and uses a Horn clause solver. Threader can detect safety violations and termination properties and is not limited to specific synchronisation primitives.

Predator and PredatorHP [PŠV20; DPV13] are tools for checking memory safety of sequential C programs. Predator uses shape analysis; i.e., it describes the shape of heap memory by an abstract domain (*symbolic memory graphs*). It targets memory errors and assertions and can handle unbounded lists (both doubly- and singly-linked, circular, and nested lists). Predator has only a limited ability to handle non-list types. To avoid parsing C directly, Predator uses an intermediate representation of the GCC compiler. PredatorHP is a tool which runs several configurations of Predator in parallel (a verifier which can prove correctness but over-approximates and several bug hunters which can show incorrectness).

SatAbs [CKSY05] is a tool for bit-precise analysis of (possibly parallel) C programs. It uses predicate abstraction with counterexample guided refinement to decrease the size of the analysed program – the program is abstracted into a boolean program which is analysed by a backend model checker. If the backend model checker concludes the abstracted program is correct, then the original program is correct. Otherwise, the trace of the abstract program is validated using SAT-queries that simulate the corresponding path of the original program. If the counterexample is spurious, SatAbs refines the abstraction. SatAbs uses bitvectors (bit-blasted to SAT formulas¹¹) and therefore is bit-precise. It can perform reachability and equivalence checking of a C program and the corresponding Verilog circuit.

An unnamed tool that encodes the behaviour of relaxed memory models in a program is presented in [AKNT13]. Given a C program which is supposed to run under one of the supported relaxed memory model (including x86-TSO and a fragment of POWER) it transforms it into an equivalent program which can be analysed by an analyser for sequentially consistent parallel programs. The authors evaluate their approach with various backend analysers, including CBMC, ESBMC, and SatAbs. The encoding of relaxed behaviour uses an *abstract event graph* that encodes events of the program and dependencies between them.

Chapter 4

Improvements in Analysis of Realistic Programs

Text of this chapter is in part extension of [ŠRB17]. The text was brought up to date with certain implementation details of DIVINE and extended with the wider context of analysis of realistic programs in DIVINE. The evaluation is unmodified from the original paper.

[ŠRB17] Štill et al., “Using Off-the-Shelf Exception Support Components in C++ Verification”.

An essential step toward the adoption of formal methods in software development is support for mainstream programming languages. These languages are often rather complex and come with substantial standard libraries. However, by choosing a suitable intermediate language, most of the complexity can be delegated to existing execution-oriented (as opposed to verification-oriented) compiler frontends and standard library implementations. In this chapter, we describe how support for C++ exceptions in DIVINE can be done by using these principles and how they are applied to support of C and C++ and their standard libraries.

As we have already outlined in Section 1.1.1, realistic programs have many features not present in verification-centric modelling languages. For example, C has dynamic memory, pointer arithmetic,¹ and function pointers. Languages with support for object-oriented programming like C++, C#, and Java have inheritance and dynamic dispatch of member functions based on the runtime type of the object they are called on. Many languages also have exceptions which allow for a non-local transfer of control between functions. C++ is particularly complex, with features like function and class templates,² inheritance with multiple base classes, runtime type information (RTTI), compiler-generated special member functions,³ and support for threads and thread synchronisation including low-level atomic access. Furthermore, many of these languages are also under constant development. For example, C++ releases a new standard every three years since 2011, with major new language and library features added in each new revision. C++ compilers and standard libraries that keep up-to-date with the standard are usually developed either by big companies like Microsoft or Intel or by community efforts like GCC and clang (with clang being also supported by Apple).

¹ Using pointer arithmetic it is possible to access elements of arrays and data structures by numeric manipulation of pointer values.

² Templates allow the programmer to write a generic code which does not depend on concrete types, the compiler is then responsible for the generation of implementations for concrete types the code is used with.

³ C++ compiler can auto-generate code that handles copying, creation and destruction of objects, and since C++20 also comparison of objects. Of course, the programmer can override the default behaviour if it is not sufficient, for example, when implementing resource abstractions.

Considering the efforts put into compiler development and the complexity of programming languages like C++, it makes sense to reuse as much of the existing execution-oriented infrastructure which exists around these programming languages. Re-implementing this infrastructure could easily consume a large portion of the limited resources developers of verification tools have. For example, DIVINE and many other tools for analysis of C and C++ use the clang compiler to translate C and C++ into LLVM intermediate representation. A similar approach can be applied to any language that uses LLVM in its translation (e.g., Rust, Swift, Objective-C), to Java (by translating it to Java Virtual Machine bytecode), or to C# (by translating it to Common Language Infrastructure). For libraries, existing open-source versions can also be used for verification to some extent.

The rest of this chapter is structured as follows: Section 4.1 takes a deeper look at the advantages and disadvantages of reuse of execution-oriented components for program verification. Section 4.2 then describes how DIVINE uses existing components in the analysis of C and C++ and why the C standard library cannot be fully reused from execution-oriented implementations. Section 4.3 then takes a more detailed look at the particular case of exception support in DIVINE 4; this part was published originally in [ŠRB17].

[ŠRB17] Štill et al., “Using Off-the-Shelf Exception Support Components in C++ Verification”.

4.1 Component Reuse in Program Analysis

Reuse of execution-oriented components (such as compilers and libraries) for program analysis is a compromise, and it is important to know the advantages and disadvantages of component reuse. On the one hand, reuse can save development time and offload the significant cost of development and testing to third parties and therefore enable support for the whole complexity of the given programming language, or at least its large parts. This is important as the whole reason for analysis of realistic programming languages is to enable the programmer to use the tool for analysis of the code they usually write, not a code written in a small subset of their language of choice. Furthermore, reuse can also increase the precision of program analysis for the cases where the same components are used for analysis and execution. For example, if the same optimising compiler is used in both situations, the analysis tool can find problems introduced by the compiler’s optimiser (which is quite complex and error-prone) that were not in the original code. On the other hand, potential problems with the reused components can also impact verification precision negatively, especially if they can hide problems in the analysed code. Even if there were no bugs in the reused components and they adhered to the specification, the problem is that these components can under-approximate all behaviours allowed by the standard. The standard can define multiple allowed ways in which a particular feature can behave, and the implementors of the library or compiler are free to choose from these options. For example, in C++, the order of evaluation of arguments of a function is not specified by the standard (and the program should not assume any particular order). The execution-oriented components are also usually

performance-oriented, which might make the task of program analysis harder.

4.1.1 Parsing and Compilation

Compilers are now routinely reused in verifiers, and the LLVM toolchain and clang compiler are especially popular in the area of C (and C++) analysis. Many program analysis tools are in fact analysing LLVM IR and not C/C++ directly (for example DIVINE [Bar+17], Skink [Cas+17], VVT [GLW16], Symbiotic [CSV18], and many more). Building program analysis for LLVM IR is simpler since LLVM IR abstracts from many of the high-level concepts and presents only a reasonable amount of instructions which has to be implemented. The clang compiler (which can compile C and C++ to LLVM IR) can handle many features of these programming languages in such a way that the program analysis tools do not need to know about them. For example, template instantiation, function overloading, dynamic dispatch, and local variable lifetime (in the absence of exceptions) are handled in this way. The compiler also handles the problem of parsing the rich grammar of languages such as C++ and translates various control-flow constructs to simple jumps.

One of the disadvantages of this approach is that the analysis tool loses some information present in the original code. For example, a loop condition is often more clearly represented in the original code than in the LLVM IR, where it has to be extracted from the value used in the conditional branch. Another disadvantage is the loss of possible behaviours – for example, the C++ standards leaves the order of evaluation of function argument unspecified, letting the compiler to optimise the code by evaluating them in any order necessary. However, on the level of LLVM IR, the evaluation order is already fixed.

Some of these disadvantages can be mitigated by reusing less of the compilation infrastructure, for example by using an approach similar to the one taken by ESBMC [Gad+18], which uses only the C and C++ parser from clang and builds their intermediate representation from the abstract syntax tree. Nevertheless, there is a cost to pay for increased precision. For example, ESBMC has to be able to handle many of the language features which are abstracted away in the compilation.

It is also possible to go in the other direction and compile the program to an assembly language for some platform before the analysis, or even to an executable binary. The disadvantage of this approach is that assembly languages often contain many details of the platform that are irrelevant for the analysis. For example, the number of registers is limited, and on some platforms, it might be hard to distinguish between function calls and jumps. In an executable, there are further complications related to address relocations and dynamic linking. Some of the advantages of this approach are that it might be easier to support multiple programming languages and that it might be possible to run the analysis on the exact same binary that is executed in production.

[Bar+17] Baranová et al., “Model Checking of C and C++ with DIVINE 4”.

[Cas+17] Cassez et al., “Skink: Static Analysis of Programs in LLVM Intermediate Representation”.

[GLW16] Günther et al., “Vienna Verification Tool: IC3 for Parallel Software”.

[CSV18] Chalupa et al., “Joint Forces for Memory Safety Checking”.

[Gad+18] Gadelha et al., “ESBMC 5.0: An Industrial-Strength C Model Checker”.

4.1.2 Libraries

Programmers expect not only the language features but also the standard libraries to work in the program analyser. These standard libraries provide fundamental features such as the data structures, algorithms, access to the file system, threading support, and memory allocation.

Clearly, any program analysis tool for C which has support for dynamic memory will need to support at least the parts of the standard library that allow its management (`malloc`, `free`, ...) and any tool that supports threading will need the support for thread spawning. These parts of the library will most likely be specific to the analysis tool, which in this context plays the same role as the operating system in the program execution. Nevertheless, large parts of the C standard library and almost all of the C++ standard library is independent of the operating system it executes on and by extension is also independent on whether it is used in the context of execution or program analysis.

The advantages and disadvantages of library reuse are in part similar to the once mentioned for compilers. The library is already tested and developed by an external entity, but it can contain bugs, or it may use design choices which are a correct implementation of the standard but limit possible behaviours of the program to a subset of behaviours allowed by the standard. However, there are some compromises specific to library reuse. Many of the bugs potentially present in a library can be caught if the library is treated by the analysis tool as a part of the analysed program. In this way, reuse of libraries can lead to the discovery of bugs in the library and therefore help to ensure correctness of the code which uses the library.⁴ The same feature also makes library reuse preferable to implementation of given features in the verifier itself – a bug in the verifier can go unnoticed, while a bug in the library has much higher chance to be caught by the verifier. Library reuse also keeps the verifier itself as small as possible, making it easier to ensure its correctness. Sadly, the same aspect of library reuse increases the complexity of the verification task, as the analysis tool now has to analyse the code of the library itself. Furthermore, production-ready libraries are usually performance-tuned, and therefore their code is more complex to analyse than a simple library implementation.

An important point in library reuse is that libraries often take full advantage of the available language features. Therefore, good support for the language is usually a prerequisite for library reuse.

It is also essential to consider what degree of library reuse makes sense. For example, a C standard library uses low-level operating system API (such as `sbrk` and `mmap` on POSIX) to implement memory allocation. It would be possible to reuse the implementation of allocation from a C standard library, provided the verifier supported this API. However, it is probably not a good idea. The OS-level API provides large blocks of memory that are subdivided by the allocation functions in the library. This subdivision (and block reuse) makes it impossible to discern boundaries of different allocations and to detect if memory is allocated or freed without detailed knowledge of the allocation algorithm. If memory is instead provided in a separate unit for each allocation, it

⁴ An example is a data race in initialization of C++ threads in `libc++` discovered by DIVINE.

The `libc++` developers later fixed this bug.

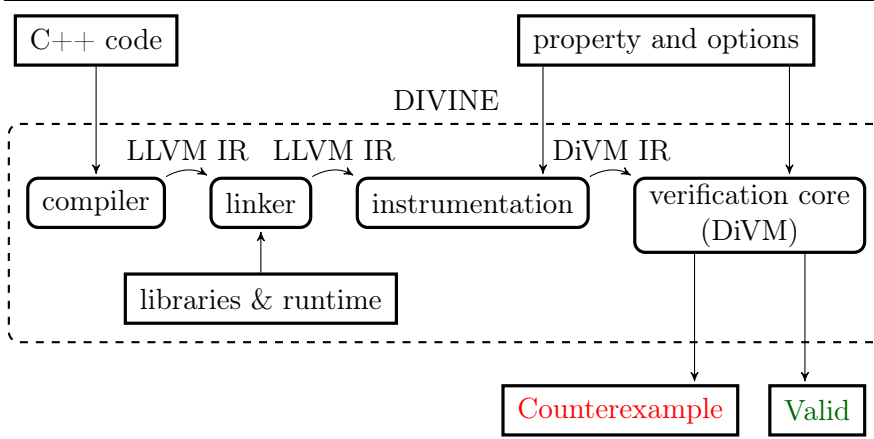


Figure 4.1: Workflow of processing of a C++ program by DIVINE. Boxes with rounded corners represent stages of input processing.

is relatively easy to detect out-of-bounds accesses and access to freed memory.

Overall, we believe that reuse of (parts of) existing library implementations is crucial for good language support. To the best of our knowledge, there are no program analysis tools which reimplement libraries which have complete or nearly complete support of the respective programming languages. Examples of these tools are ES-BMC++ [Ram+13] for C++, SMACK for C, C++, Rust and many other [GBHR20], and Java Pathfinder [AV19] for Java, all of which mention that they support only parts of the respective standard libraries.

4.2 Component Reuse in DIVINE

In DIVINE, we use the clang compiler to produce LLVM IR. The IR is then linked with implementations of libraries and instrumented for the verification. The LLVM IR with verification-specific instrumentation will be called DiVM IR, matching the naming introduced by [Bar+17]. DiVM is the core of DIVINE which handles the execution of LLVM in a memory-safe way and can save and restore state of the execution. An overview of the processing of C++ code in DIVINE can be seen in Figure 4.1.

To make the compilation as easy as possible, the clang compiler is integrated into DIVINE as a library, and the libraries which are linked with the program are built at the time of build of DIVINE. The original integration of clang into DIVINE 4 and the adaptation of LLVM linker for use with DIVINE was done by me. The work was then taken over by other members of the team. The current state of the compilation for DIVINE is described in [BR20].

To provide support for most of C and C++, DIVINE has to provide standard libraries for these languages. DIVINE has support for the C and C++ standard libraries, and for the POSIX thread library (`pthread`), which is used on POSIX systems for implementation of threading in C/C++ code before the 2011 standards and implementation of the standard threading in the later versions of C/C++. DIVINE

[Ram+13] Ramalho et al., “SMT-Based Bounded Model Checking of C++ Programs”.

[GBHR20] Garzella et al., “Leveraging Compiler Intermediate Representation for Multi- and Cross-Language Verification”.

[AV19] Artho et al., “Java Pathfinder at SV-COMP 2019 (Competition Contribution)”.

[Bar+17] Baranová et al., “Model Checking of C and C++ with DIVINE 4”.

[BR20] Baranová et al., “Compiling C and C++ Programs for Dynamic White-Box Analysis”.

[Roč+19] Ročkai et al., “Reproducible Execution of POSIX Programs with DiOS”.

also has support for many POSIX standard functions, including large parts of the file system API [Roč+19].

The threading library is written specifically for verification with DIVINE. Indeed, the whole idea of verification of parallel programs requires that at some level of the abstraction, a verification-specific semantics of threads is given. In the case of DIVINE, DiVM provides explicit nondeterministic choice primitives which are used by the core of DIVINE’s runtime (DiOS) to implement thread spawning, which is in turn used by the implementation of POSIX thread library to start threads. Mutexes, condition variables, and other synchronisation primitives, which in execution environment usually use an operating system for the waiting and signalling, are implemented using atomic sections.

Similarly to threading support, parts of the C standard library have to be implemented specifically for the verifier. In our case, these parts are mostly memory allocation and deallocation, which takes memory directly from DiVM, program startup and exit which is DiOS specific, and handling of the `errno` variable which is also DiOS specific. The rest of the standard C library originates mostly from PDClib, with some parts from various other open-source C library implementations and contributions from DIVINE developers.

The C++ standard library usually builds on top of a C standard library. This allowed us to reuse existing implementation of the C++ standard library, namely `libc++abi` (which takes care of low-level features such as memory allocation and runtime type support, which can be viewed as part of the language itself but need a library implementation in practice) and `libc++` (which represents all the user-facing features of the C++ standard library). Both of these libraries are part of the LLVM project and used for example on macOS and certain BSD flavours as the default C++ library implementations. Reuse of these libraries takes care of most of the C++ features which require library support, with the sole exception of exceptions, which require a special library for stack manipulation – an *unwinder*.

The unwinder library used in DIVINE is DiVM specific and is described, together with more details about exception support in DIVINE in the next section. Nevertheless, by adhering to the interface of the unwinder library used typically on POSIX systems, we were able to make use of all the exception handling code in `libc++abi` without any changes to it. The changes in the source code of `libc++` and `libc++abi` are limited to changes to platform configuration macros, and in `libc++abi` there is a change in the allocation of thread-local backup storage for exception handling.

The POSIX APIs, mainly concerning file system, are provided by DiOS and were implemented for use with DIVINE or other program analysis tools [Roč+19]. This functionality makes it possible to analyze programs which use file system and other supported parts of POSIX, either with a simulated environment or with recording and replay of interaction with the real environment.

The library support and the situation concerning library reuse in DIVINE are summarized by Table 4.1.

Library	Reused?	Comment
C standard	mostly (PDClib, other)	Platform dependent code such as allocation, program startup and exit, errno cannot be reused without replicating too many details of the operating system.
C++ standard	yes (libc++)	Only changes to platform selection macros needed.
C++ runtime	yes (libc++abi)	Platform selection + tweak of backup allocator for exceptions (to prevent performance penalty).
POSIX threads	no	Needs to be verification specific to allow exploration of all behaviours of a parallel program.
POSIX filesystem	no	Normally requires operating system support and cannot revert into previous state. Verified program should not access outside environment directly.
stack unwinder	no	Platform specific (depends on stack layout and metadata format), cannot be reused without replicating too many details of the hardware platform.

Table 4.1: Summary of reuse and reimplementation of libraries in DIVINE.

The libraries used for verification are shipped with DIVINE source code and compiled into LLVM IR at the time of compilation of DIVINE. These LLVM IR libraries are then linked to the program at the time of its compilation for DIVINE. The verification workflow and the execution workflow are rather similar in the case of DIVINE (especially on POSIX systems). In both cases, the program is compiled and then linked with implementation of C and C++ standard libraries, POSIX thread library, and a stack unwinder.

4.3 C++ Exceptions in DIVINE

Exceptions in C++ are among the features that are both widely used (also by the standard library) and tricky to implement. Their use is, however, also common outside of the standard library: libraries like boost⁵ and application-level code often takes advantage of this capability. This is natural, since exceptions simplify error handling and usually require less boilerplate code than any of the alternatives. Furthermore, even though many C++ standard library implementations can be built without exception support⁶, this change can significantly affect its behaviour (and as such, validity of the verification result).

⁵ Boost is one of the most used collections of general-purpose C++ libraries. Many features of boost eventually get into the C++ standard.

⁶ There are cases where not using exceptions makes sense: if the end-user code makes no use of them but the standard library is compiled with exception support, the requisite metadata tables only serve to increase the size of the compiled program.

Finally, error handling paths, including exception propagation, are an important target for analysis by verification tools, as they are both hard to test by more conventional means and likely to contain errors – this naturally arises from the fact that their purpose is to handle unlikely side cases which can be hard to accurately reproduce with testing. A model checker, on the other hand, can take advantage of its built-in support for nondeterminism to rigorously explore error paths.⁷

4.3.1 Contribution

The approach to exception support described in this section, and originally published in [ŠRB17] brings the following contributions: first, we identify the components that are best reused and those which are best reimplemented and show that this decision crucially depends on the underlying intermediate language. Second, we provide implementations of the components which cannot be reused in a form that is easy to integrate into both existing and future verification tools. One of the components works as an LLVM transformation pass, and could be used with any LLVM-based tool. The other component targets the DiVM language [RŠČB18] specifically, and will therefore only work with tools which understand this language.⁸

The goal of this work, especially in the context of our previous work on the topic of C++ exceptions in verification [RBB16], is to aid authors of verification tools to minimise costs and effort associated with inclusion of exception support. Depending on the characteristics of the tool, either the approach described in [RBB16] or the one in this work might be more suitable. Overall, in a verifier which can handle the DiVM language or equivalent, the approach given in this chapter is simpler to implement and more robust. A more detailed comparison of the two approaches is given in Section 4.9.2.

All source code related to this work, along with supplementary material, is available online under a permissive open-source licence.⁹ The implementation is also a fully integrated part of the DIVINE model checker and therefore present in its current versions.

4.3.2 Components for Exception Support

Unlike other features of C++, exceptions are neither handled by the standard or runtime libraries alone, nor delegated to the C standard library (as C has no support for exceptions). Instead, `libc++abi` provides exception support with the help of a platform-specific *unwinder library* which is responsible for stack introspection and unwinding (removal of stack frames and transfer of control to exception-handling code).

For this reason, DIVINE has to either provide an unwinder implementation compatible with `libc++abi`, or modify `libc++abi` to use custom code for exception handling. In DIVINE 3, the latter approach was used, as it was deemed easier at the time [RBB16]. However, while basic exception support was easier to achieve this way, the approach also had its disadvantages. First, the LLVM interpreter in DIVINE 3 had special support for exception-related functionality. Second, the

⁷ This is a form of fault injection. When using a model checker, it is only necessary to modify the function where the error may arise (e.g. the `malloc` function may be modified to return a `NULL` pointer nondeterministically). The model checker will then take care of exploring all possible combinations of succeeding and failing memory allocations in the program.

[ŠRB17] Štill et al., “Using Off-the-Shelf Exception Support Components in C++ Verification”.

[RŠČB18] Ročkai et al., “DiVM: Model checking with LLVM and graph memory”.

⁸ DiVM is a relatively small extension of the LLVM IR, therefore extending tools which work with pure LLVM to also support DiVM may be quite easy.

[RBB16] Ročkai et al., “Model Checking C++ Programs with Exceptions”.

⁹ <https://divine.fi.muni.cz/2017/exceptions>

libc++abi code for exception handling was replaced, which had two important consequences: first, the replacement code was not comprehensive enough¹⁰ and second, this meant that the replaced part of libc++abi was not taken into account during verification.

In this work, we instead take the first approach: reuse libc++abi in its entirety and provide the interfaces it requires. Therefore, we have implemented the *libunwind* interface used by libc++abi for stack unwinding and an LLVM instrumentation which builds metadata tables that libc++abi needs to decide which exceptions should be caught, how they should be handled and which functions on the stack need to perform cleanup actions. To put this into perspective, the code which drives exception handling in libc++abi is about 1300 lines of code, while the code related to libunwind implementation in DIVINE is about 210 lines and the instrumentation for C++ specific metadata is about 300 lines of code.

Using the original libc++abi code means that all features of the C++ exception system are fully supported and verification results also cover the low-level exception support code. That is, this portion of the code is identical in both the bitcode which is verified and in the natively executing program.¹¹ Furthermore, should the need arise to use different C++ runtime library then libc++abi (for example to provide additional guarantees for programs which use this different library), our solution should work without any modification of the given C++ runtime library, provided it uses the (semi-standard) unwinder interface. Finally, the proposed design is easier to extend to other programming languages as the libunwind implementation is generic and language independent and only the instrumentation which provides language-specific metadata needs to be provided to different languages.

4.4 Exceptions in C++

The process of exception handling in C++ is illustrated by Figure 4.2.

Throwing an exception requires removal of all the stack frames¹² between the throwing and catching function from the stack (*unwinding*). Therefore, exception handling is closely tied to the particular platform and is described by ABI¹³ for the platform. Commonly, exception handling is split into two parts, one which is tied to the platform (the *unwinder library* which handles stack unwinding) and one which is tied to the language and provided by the language's runtime library.¹⁴ These two parts cooperate in order to provide exception handling for a given language; however, this communication is not standardised in any cross-platform fashion. For this reason, we will now focus on zero-cost exceptions based on the Itanium ABI, an approach which is used across various Unix-like systems on x86 and x86-64 processor architectures and is the preferred basis for LLVM exceptions. Nevertheless, it is possible to generalize our results to other implementations.

4.4.1 Zero-Cost Exceptions

The so-called zero-cost exceptions are designed to incur no overhead during normal execution, at the expense of relatively costly mechanism

¹⁰ That is, some of the less frequently used features of C++ exceptions were handled either incorrectly or not at all. That is to say, the size of the libc++abi portion that would have needed to be reimplemented was initially underestimated.

¹¹ Clearly, the libunwind implementation is different in those two environments, and therefore correctness of the platform-specific implementation of libunwind must be established separately.

¹² The execution stack of a (C++) program consists of stack frames, each holding context of a single entry into some function. It contains local variables, a return address and register values which need to be restored upon return.

¹³ *Application Binary Interface*, a low-level interface between program components on a given platform.

¹⁴ There are many implementations of the C++ runtime library, which, besides exception support code, provides additional features such as RTTI. Each implementation is usually tied to a particular implementation of the C++ standard library. Commonly used implementations on Unix-like systems are libsupc++, which comes with libstdc++ and the GCC compiler, and libc++abi, which is tied to libc++ used by some builds of clang and by DIVINE.

```

int max(std::vector<int> &vec) {
    if (vec.empty())
        throw std::invalid_argument("empty vector");
    /* ... */
}

void foo() {
    std::vector<int> a = { 1, 2, 3 };
    std::vector<int> b = {};
    auto x = max(b);
    /* ... */
}

int main() {
    try {
        foo();
    } catch (std::range_error &) {
        std::cout << "range error\n";
    } catch (std::exception &) {
        std::cout << "other exception\n";
    }
}

```

Figure 4.2: Exception handling in C++. At the moment an execution is thrown in `max`, there are three frames on the stack `max`, `foo`, and `main`. The exception can be caught by `main`, but first the cleanup code in `foo` has to be executed – this code will deallocate memory owned by the vectors present in this function. Therefore, first only the stack frame of `max` is removed and the cleanup in `foo` is executed. After the cleanup, the stack frame of `foo` will be removed from stack. Finally in `main`, the unwinder will transfer control to the second catch statement (`std::exception` is predecessor of `std::invalid_argument`, but `std::range_error` is not its predecessor).

for throwing exceptions. This in particular means that no checkpointing is possible. Instead, when an exception is thrown, the exception support library, with the help of *unwind tables*, finds an appropriate *handler* for the exception and uses the *unwinder* to manipulate the stack so that this handler can be executed. The search for the handler is driven by a *personality function*, which is provided by the implementation of the particular programming language and associated in the metadata with each function which can participate in exception handling.

The personality function is responsible for deciding which handler should execute (the handler selection can be complex and language-specific). In general, there are two types of handlers, *cleanup handlers*, which are used to clean up lexically scoped variables (and call their destructors, as appropriate) and *catch handlers*, which contain dedicated exception-handling code. The latter typically arise from `catch` blocks. Another major difference between those two types of handlers is that catch handlers stop the propagation of the exception, while cleanup handlers let propagation continue after the cleanup is performed. While cleanup handlers are usually run unconditionally, the catch handler

to be executed, if any, is determined by the personality function.¹⁵ In C++, the personality function selects the closest `catch` statement which matches the thrown type (the match is determined dynamically, using RTTI). The personality function consults the unwind tables, in particular their *language-specific data area (LSDA)*, to find information about the relevant catch handlers.

When an exception is thrown, the runtime library of the language creates an *exception object* and passes it to the unwinder library. The actual stack unwinding is, on platforms which build on the Itanium ABI, performed in two phases (*two-phase handler lookup*). First, the stack is inspected (without modification) in search for a catch handler. Each stack frame is examined by the relevant personality function.¹⁶ If an appropriate catch handler is found in this phase, unwinding continues with a second phase; otherwise, an unwinder error is reported back to the throwing function. Unwinder errors usually cause program termination. In the second phase, the stack is examined again, and a personality function is invoked again for each frame. In this phase, cleanup handlers come into play. If any handler is found (cleanup or catch), this fact is indicated to the unwinder, which performs the actual unwinding to the flagged frame. Once the control is transferred to the handler, it can either perform cleanup and resume propagation of the exception, or, if it is a catch handler, end the propagation of the exception. If exception propagation is resumed, the unwinder continues performing phase 2 from the point of the last executed handler. This is facilitated by storing the state of the unwinder within the exception object.

4.4.2 Unwind Tables

As mentioned in Section 4.4.1, both the unwinder library and the language runtime depend on unwind tables for their work. The unwinder uses these tables to get information about stack layout in order to be able to unwind frames from it, and for detection which personality function corresponds to a frame. The personality function then uses the language-specific data area (LSDA) of these tables in its decision process.

While the unwinder part of the tables is unwinder- and platform-specific (it depends on stack layout), the LSDA is platform- and language-specific. For these reasons, unwind tables are not present in the LLVM IR; instead, they are generated by the appropriate code generator for any given platform, based on information in the `landingpad` instructions, and the personality attribute of functions. On Unix-like systems, the unwind tables are in the DWARF¹⁷ format.

4.5 Execution of LLVM programs

In this section, we will look at how LLVM bitcode is executed by a model checker and how this execution is affected by addition of exception support. Unlike previous approaches, the technique described in this chapter does not require any exception-specific intrinsic functions or hypercalls to be supported by the verifier. The exception-specific LLVM

¹⁵ In fact, the personality function can also decide to skip cleanup handlers, but this is not common.

¹⁶ Different personality functions can be called for different frames, for example if the program consists of code written in different languages with exception support.

¹⁷ DWARF is a standard for debugging information designed for use with ELF executables. It is used on most modern Unix-like systems.

instructions can be implemented in the simplest possible way: `invoke` becomes equivalent to a `call` instruction followed by an unconditional branch. The `landingpad` instruction can be simply ignored by the verifier and `resume` instructions and calls of the `llvm.eh.typeid.for` intrinsic are both removed by the LLVM transformation and instrumentation described in Section 4.6. Moreover, the metadata required by `libc++abi` is likewise generated by the LLVM transformation and this process is completely transparent to the verifier.

In addition to support for LLVM, the unwinder (described in more detail in Section 4.7) requires the ability to traverse and manipulate the stack and read and write LLVM registers associated with a given stack frame. Finally, it needs access to a representation of the bitcode for a given function. All those abilities are part of the DiVM specification [RŠČB18] and are generally useful, regardless of their role in exception support.

The DiVM implementation in DIVINE 4 handles execution of LLVM instructions, LLVM intrinsic functions and DiVM-specific *hypercalls*.¹⁸ Hypercalls exist to allocate memory, perform nondeterministic choice or to set DiVM’s *control registers* (which contain, among other, the pointer to the currently executing stack frame). Additionally, DiVM performs safety checks, such as memory bound checking, and detects use of uninitialised values. However, DiVM hypercalls are intentionally low-level and simple and do not provide any high-level functionality, such as threading or standard C library functionality. Instead, those are provided by the DIVINE Operating System (DiOS) and the regular C and C++ standard libraries.

The most important purpose of DiOS is to provide threading support. To this end, DiOS provides a *scheduler*, which is responsible for keeping track of threads and their stacks and for (nondeterministically) deciding which thread should execute next. This scheduler is invoked repeatedly by the verifier to construct the state space. The scheduler fully determines the behaviour (or even presence) of concurrency in the verified program: while DiOS provides asynchronous, preemptive parallelism typical of modern operating systems by default, it has also support for a synchronous scheduler and it would be possible to implement a cooperative scheduler too.

4.5.1 Stack Layout and Control Registers

A DiVM program can have multiple stacks, but only one of them can be active at any given time (a pointer to the active stack is kept in a DiVM control register). The active stack is normally either the kernel stack or the stack that belongs to the active thread which was selected by the scheduler. Switching of stacks (and program counters) is performed by the `control` hypercall which manipulates DiVM control registers.

Traditionally, stack is represented as a contiguous block of memory which contains an activation frame for each function call. In DiVM, the stack is not contiguous; instead, it is a singly-linked list of activation frames, each of which points to its caller. This has multiple advantages: first, it is easy to create a stack frame for a function, for example when DiOS needs to create a new thread; additionally, the linked-

[RŠČB18] Ročkai et al., “DiVM: Model checking with LLVM and graph memory”.

¹⁸ Intrinsic functions are provided by LLVM as a light-weight alternative to new instructions. Such functions are recognized and translated by LLVM itself, as opposed to “normal” functions that come from libraries or the program. Likewise, DiVM provides hypercalls, which are functions that are, in addition to LLVM intrinsics, recognized by DiVM.

list-organized stack is a natural match for the graph representation of memory which DiVM mandates, and therefore can be saved more efficiently [RŠČB18]. Additionally, this way the stack may be nonlinear, and the unwinder can use this feature to safely transfer control to a cleanup block while the unwinder frame is still on the stack. Later, the handler can return control to the unwinder frame and the unwinder can continue its execution. This would be impossible with a contiguous stack since cleanup code is allowed to call arbitrary functions and frames of those functions would overwrite the frame of the unwinder. For this reason, on traditional platforms, the unwinder needs to store its entire state in the exception object, while in DiVM, it can simply retain its own activation frame, which simplifies the unwinder. An illustration of how the stack looks while the unwinder is active is shown in Figure 4.3.

[RŠČB18] Ročkait et al., “DiVM: Model checking with LLVM and graph memory”.

4.6 The LLVM Transformation

The C++ runtime library (`libc++abi` in our case), needs access to the LSDA section of unwind tables (a pointer to this metadata section is accessible through the unwinder interface). This section contains DWARF-encoded exception tables, which are normally generated together with the executable by the compiler backend (code generator). Unfortunately, the generator of DWARF exception tables in LLVM is closely tied to the machine code generator and cannot be used to generate DWARF-formatted exception tables for the LLVM IR used for verification purposes. For this reason, we have implemented a small LLVM transformation which processes the information in `landingpad` instructions and generates LLVM constants which contain the DWARF-formatted LSDA data. A reference to one such constant is attached to each function in the bitcode file.

The choice of catch block in C++ depends on the actual type the exception has at runtime. This type can be inspected thanks to runtime type information (RTTI). The RTTI can also be used to determine if one class inherits from other class, which is needed to decide which catch block can be used (as catch blocks can catch exceptions of derived types). The RTTI is available through RTTI type info pointers, special C++ objects which are used to identify types at runtime and are emitted by the C++ frontend as constants. Type info pointers can be obtained from an object using the `typeid` operator in C++ and in LLVM they are saved in the virtual member functions’ table (*vtable*; for objects which have virtual member functions, other objects do not have RTTI type info pointers saved in them and the corresponding type info is fully determined by their static type).

To improve efficiency, LLVM and zero-cost exceptions do not directly use RTTI type info pointers within the landing blocks to decide which exception handlers should run. Due to the complexities of C++ type system, matching RTTI types against each other is expensive: a search in a pair of directed acyclic graphs is required. Moreover, since the RTTI matching must be already done in the personality function to decide which frames to unwind, the personality can also pre-compute a numerical index for the landing pad. This index, also called a *selector*

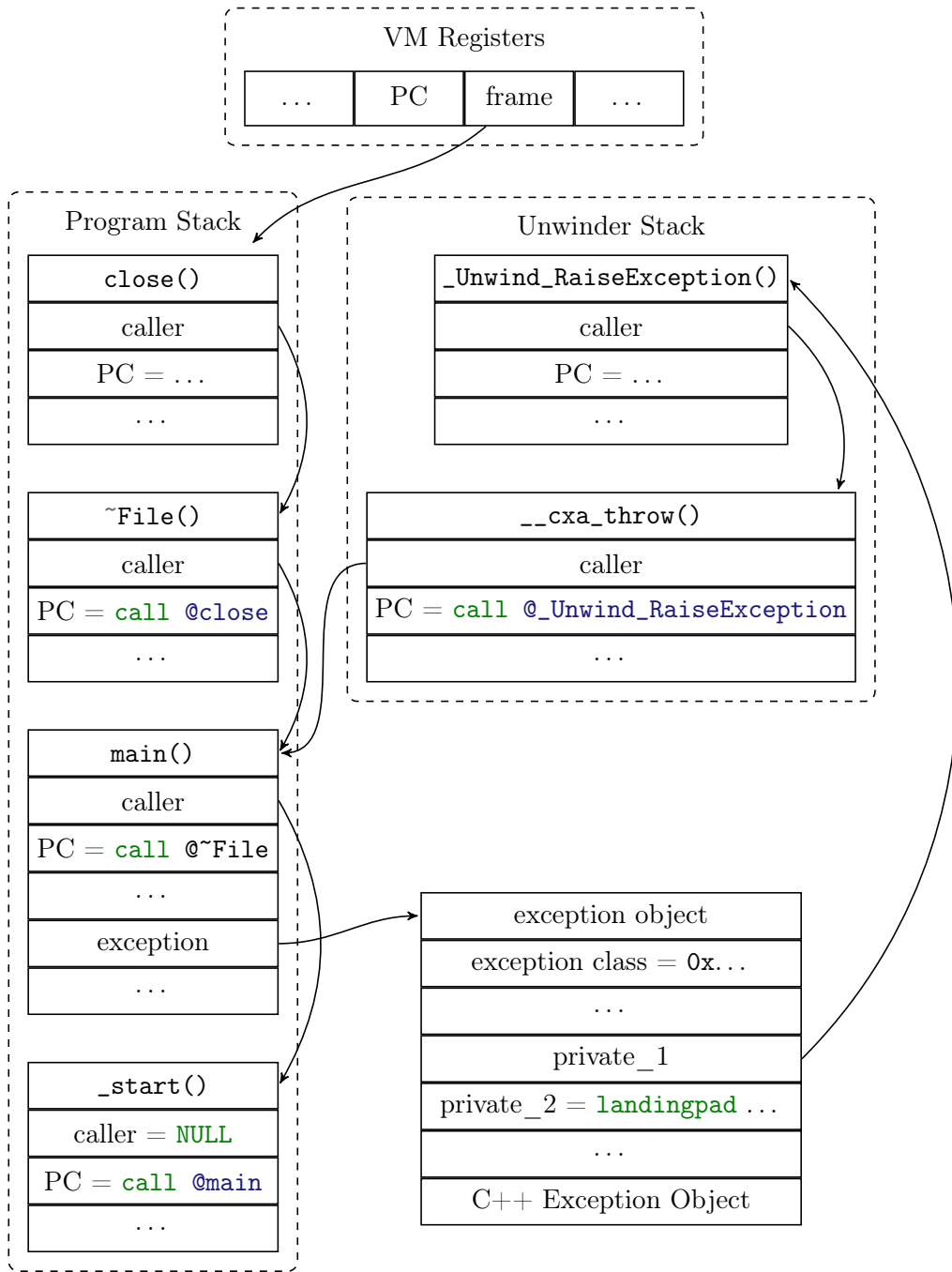


Figure 4.3: In this figure, we can see a stack of a program which is running cleanup block in the `main` function. The cleanup block calls the destructor of `File` structure, which in turn calls the `close` function (which is the current active function). Furthermore, the cleanup handler can access the exception object which contains a pointer to the stack of the unwinder (and the program counter of the catch block). The frame pointer in the exception is used by the implementation of the `resume` instruction to jump back to the unwinder and continue phase 2 of the unwinding.

Table 4.2: A list of C functions provided by `libunwind`. In C, all the functions are prefixed with `_Unwind_` to prevent name conflicts with user code and other libraries (i.e. the C name of `SetGR` is `_Unwind_SetGR`).

Function	Description
<code>SetGR</code>	Store a value into a general-purpose register
<code>GetGR</code>	Read a value from a general-purpose register
<code>SetIP</code>	Store a value into the program counter
<code>GetIP</code>	Read the value of the program counter
<code>RaiseException</code>	Unwind the stack
<code>Resume</code>	Continue unwinding the stack after a cleanup
<code>DeleteException</code>	Delete an exception object (free its resources)
<code>GetLSDA</code>	Obtain a pointer to the LSDA
<code>GetRegionStart</code>	Obtain a base for relative code pointers

value is then used as a shortcut to run an appropriate `catch` clause within the landing block, instead of re-doing the expensive RTTI matching. Since the `catch` handler is typically expressed in terms of type info pointers, it needs to efficiently obtain the selector value from a type info pointer. For this purpose, LLVM provides a `llvm.eh.typeid.for` intrinsic, which obtains (preferably at compile time) the selector value corresponding to a particular type info pointer.

Therefore, besides generating the LSDA data, the transformation statically computes the values of calls to `llvm.eh.typeid.for` and substitutes them into the bytecode. The purpose of `llvm.eh.typeid.for` is to translate from RTTI pointers to selector values, therefore it is only required that the integer selector value chosen for a particular RTTI object is in agreement with the personality function. In our implementation, this is ensured by computing the selector values statically for both the LSDA (which is where personality function obtains them) and for `llvm.eh.typeid.for` at the same time.

Finally, the transformation rewrites all uses of the `resume` instruction to ordinary calls to `Resume`, a function which is part of `libunwind` (see also Table 4.2).

4.7 The Unwinder

The unwinder in DIVINE is designed around the interface described in the Itanium C++ ABI documentation,¹⁹ adopted by multiple vendors and across multiple architectures. The implementation is part of the runtime libraries shipped with DIVINE, in particular in file `dios/arch/divm/unwind.cpp`. The unwinder builds upon a lower-level stack access API which is provided by DiOS under `dios/include/sys/stack.h` and implemented in `dios/arch/divm/stack.cpp`.

Due to the stack layout used in DiVM (a linked list of frames, see also Section 4.5.1), our unwinder is much simpler than usual. The main task of unwinding is handled by the `RaiseException` function, which is called by the language runtime when an exception is thrown. This function performs the two-phase handler lookup described in

¹⁹ <https://mentoreembedded.github.io/cxx-abi/abi.html>

Section 4.4.1 and it adheres to the Itanium ABI specification, with the following exceptions:

- i. it checks that an exception is not propagated out of a function which has the `nounwind` attribute set, and reports verification error if this is the case;
- ii. if the exception is a C++ exception and there is no handler for this exception type, the unwinder chooses nondeterministically whether it should or should not unwind the stack and invoke cleanup handlers.

The purpose of the first deviation is to check consistency of exception annotations (arising, for example, from a `nothrow` function attribute as available in GCC and in clang). The second modification allows DIVINE to check both allowed behaviours of uncaught exceptions in C++: the C++ standard specifies that it is implementation-defined whether the stack is unwound (and destructors invoked) when an exception is not caught.²⁰ Since the program may contain errors which manifest only under one of these behaviours, it is useful to be able to test both of them.

²⁰ Section 14.4, paragraph 9 of the C++20 standard [ISO20]

4.7.1 Low-Level Unwinding

The primary function of the unwinder described above is to find exception handlers; for the actual unwinding of frames, it uses a lower-level interface provided by DiOS. This interface consists of two functions: `__dios_jump`, which performs a non-local jump, possibly affecting both the program counter and the active frame, and `__dios_stack_free`, which removes stack frames from a given stack. `__dios_stack_free` is designed in such a way that it can unwind any stack, not only the one it is running on, and is not limited to the topmost frames (effectively, it removes frames from the stack's singly-linked list, freeing all the memory allocated for local variables that belong to the unlinked frames, along with the frames themselves²¹). The unwinder identifies values as local variables by looking at the instructions of the active function – the results of `alloca` instructions are exactly the addresses of local variables.

²¹ When a function returns normally (due to a `ret` instruction), DiVM takes care of freeing the frame and its local variables (`alloca` memory).

4.7.2 Unwinder Registers

When an exception is propagating, a personality function has to be able to communicate with the code which handles the exception. In C++, the communicated information includes the address of the exception object and a selector value which is later used by the handler. On most platforms, these values are passed to the handler using processor registers, which are manipulated using unwinder's `SetGR` function. This function can either set the register directly (if it is guaranteed not to be overwritten before the control is transferred to the handler), or save the value in a platform-specific way and make sure it is restored before the handler is invoked.

In LLVM (and hence in DiVM), there is no suitable counterpart to the general purpose registers of a CPU; instead, the values set by the personality function should be made available to the program in the return value of the `landingpad` instruction. This, however, requires the knowledge of the expected semantics of these registers. Currently, all users of the unwinder are expected to use the same registers as the C++ frontend in clang. That is, register 0 corresponds to the exception object and register 1 corresponds to a type index. This also directly maps to the return type of `landingpad` instructions and therefore the register values can be saved directly into the LLVM register corresponding to the particular `landingpad` that is about to be executed.²²

Registers other than 0 and 1 are currently not supported. In LLVM, in line with the above observation about clang and C++, there is a convention that `SetGR` indices correspond to indices into the result tuple of a `landingpad` instruction. As long as this convention is preserved by a particular language frontend and its corresponding runtime library (personality function), it is very easy to extend our unwinder to support this language. Finally, if a language frontend were instead to emit calls to `GetGR` in the handler, registers of this type can be stored in the unwinder context directly.

²² LLVM registers are function-local values that cannot be used to pass information between functions in pure LLVM. With DiVM, we can use meta-data to locate the register's value in a frame of a function and modify it.

4.7.3 Atomicity of the Unwinder

The unwinder performs rather complex operations and therefore throwing an exception can create many states, even when τ reduction [RBB13] is enabled. However, many of these states are not interesting from the point of view of verification, as the operations performed by the unwinder are mostly thread-local and only the exception handlers (and possibly personality function) can perform globally visible actions. For this reason, the unwinder uses DiOS atomic sections to hide most of its complexity.

Since an atomic section is implemented as an *interrupt mask* (i.e. a single flag indicating that an atomic section is executing) in DiVM's flag register, it is necessary to correctly maintain the state of this flag. In particular, it is required that the unwinder behaves reasonably even if it is called when the program is already in an atomic section. Consequently, care must be taken to restore the state of the atomic mask when the unwinder transfers control to a personality function or an exception handler. When the unwinder is first called, it enters an atomic section and saves the previous value of the interrupt mask. This will be the value the flag will be restored to when a personality function is first invoked. The mask is later re-acquired after the personality function returns and it is restored once more when the first handler is invoked. When the exception handler resumes (using the `resume` instruction), the atomic section is re-entered and its state saved so its state before the resume can be restored again for the next call to a personality function. This way, it is possible to safely throw an exception out of an atomic section, provided that the atomic section is exception-safe (that is, it has an exception handler which ends the atomic section if an exception is propagated out of it). This is a reasonable assumption because atomic sections are mostly used in the implementation of DiOS

[RBB13] Ročkaitis et al., “Improved State Space Reductions for LTL Model Checking of C & C++ Programs”.

itself, and they should use C++ guard objects to ensure exception safety.

4.7.4 longjmp Support

Using the low-level unwinder interface described in Section 4.7.1, it is easy to implement other mechanisms for non-local transfer of control. The functions `longjmp` and `setjmp`, specified as part of C89, are one such example.²³ The `setjmp` function can be used to save part of the state of the program, so that a later call to `longjmp` can restore the stack to the state it was in when `setjmp` was called. This way, `longjmp` can be used to remove multiple frames from the stack. When `longjmp` is called, the program behaves as if `setjmp` returned again, only this time it returns a different value (provided as an argument to `longjmp`).

The DIVINE implementation of `setjmp` saves the program counter and the frame pointer of the caller of `setjmp`. The `longjmp` function then uses this saved state, along with access to metadata about stack frames, to set the return value of the `call` instruction corresponding to the `setjmp`. Afterwards, it unwinds the stack using the low-level stack access API and transfers control to the instruction right after the call to `setjmp`.

4.8 Related Work

Primarily, we have looked at existing tools which support verification of C++ programs. Existence of an implementation is, to a certain degree, an indication that a given approach is viable in practice. We have, however, also looked at approaches proposed in the literature which have no implementations (or only a prototype) available.

A number of verification tools are based on LLVM and therefore have some support for C++. LLBMC [FMS13] and NBIS [GW14] are LLVM-based bounded model checkers which target mainly C and have no support for exceptions or the C++ standard library. VVT [GLW16] is a successor of NBIS which uses either IC3 or bounded model checking and has limited C++ support, but it does not support exceptions. Furthermore, KLEE [CDE08] and KLOVER [LGR11] are LLVM-based tools for test generation and symbolic execution. KLOVER targets C++ and according to [LGR11] has exception support, but it is not publicly available. On the other hand, KLEE focuses primarily on C and its C++ support is rather limited. As of April 2020, there is ongoing work on exception support in KLEE.

Both CBMC [CKL04; KT14] and ESBMC [Gad+18] bounded model checkers support C++ (but neither appears to have working support of the standard library) and they appear to have some support for exceptions. However, in CBMC, the support for exceptions seems to be limited to recognition of the `try-catch` syntax and throwing an exception triggers an error.²⁴ In our survey of tools for verification of C++ programs, ESBMC has by far the best exception support: ESBMC 3.0 can deal with most, but not all, types of exception handlers and even with exception specifications.²⁵ However, ESBMC is currently (in 2020, as of version 6.3) migrating C++ to the clang frontend and does not

²³ Implemented in `dios/includes/setjmp.h` and `dios/libc/setjmp/`.

[FMS13] Falke et al., “The bounded model checker LLBMC”.

[GW14] Günther et al., “Incremental Bounded Software Model Checking”.

[GLW16] Günther et al., “Vienna Verification Tool: IC3 for Parallel Software”.

[CDE08] Cadar et al., “KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs”.

[LGR11] Li et al., “KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs”.

[CKL04] Clarke et al., “A Tool for Checking ANSI-C Programs”.

[KT14] Kroening et al., “CBMC – C Bounded Model Checker”.

[Gad+18] Gadelha et al., “ESBMC 5.0: An Industrial-Strength C Model Checker”.

²⁴ A simple test which throws and tries to catch an exception object crashes CBCM 5.11 during the verification and a code which attempts to catch an exception by reference (which is the recommended practice in C++) results in spurious syntax error during program parsing.

support exceptions. Finally, DIVINE 3 [RBB16] also comes close to full support for exceptions, but lacks support for exception specifications. Overall, this survey suggests that all current implementations of C++ exceptions in verification tools are incomplete and confirms that using an existing, standards-compliant implementation in a verification tool is indeed quite desirable.

Finally, it is also possible to transform a C++ program with exceptions into an equivalent program which only uses more traditional control flow constructs. This approach was taken in [Pra+11], with the goal of reusing existing analysis tools without exception support. While this approach is applicable to a wide array of verification tools, it is also incompatible with reuse of existing exception-related runtime library code. As such, it offers a very different set of tradeoffs than our current approach. Moreover, the translation cost is far from negligible, and also affects code that does not directly deal with exceptions (i.e. it violates the zero-cost principle of modern exception handling). Unfortunately, we were unable to evaluate this approach, since there are no publicly available tools which would implement it.

Exceptions are more widely supported in the context of Java analysis, probably thanks to their prevalence in Java code. Examples of tools for analysis of Java with exception support are Java Pathfinder [AV19], JBMC [CKS19], and JDart [MH20].

4.9 Evaluation

In order to assess the viability of our approach, we have executed a set of benchmarks in various configurations of DIVINE 4. The benchmarks were executed on quad-core Xeon 5130 clocked at 2 GHz and with 16GB of RAM. We have measured the wall time, making all 4 cores available to the verifier.

4.9.1 Benchmark Models

The set of models we have used for this comparison consists of 831 model instances, out of which we picked the 794 that do not contain errors. The reason for this is that the execution time is much more variable when a given program contains an error, since the model checking algorithm works on the fly, stopping as soon as the error is discovered and explores the state space in parallel, which makes the exploration order vary between verification runs.

Majority of the valid models (777) are C++ programs of varying complexity, while the 17 models in the `svc-pthread` category are concurrent programs written in plain C with pthreads. Since our implementation of the POSIX thread API is done in C++, the impact of exception support on verification of C programs is also relevant. The “alg” category includes sequential algorithmic and data structure benchmarks, the “pv264” category contains unit tests for student assignments in a C++ course, the “iv112” category contains unit tests for concurrent data structures and other parallel programs (again assignment problems in a course about concurrency in C++), “libcxx” contains a selection of tests from the `libc++` testsuite (with focus on exception

²⁵ ESBMC 3.0 is unable to determine that an exception ought to be caught when the `catch` clause specifies a type which is a virtual base class in a diamond-shaped hierarchy and the object thrown is of the most-derived type of the diamond. This suggests that ESBMC uses its own implementation of RTTI support code, which is somewhat incomplete, compared to production implementations.

[RBB16] Ročkai et al., “Model Checking C++ Programs with Exceptions”.

[Pra+11] Prabhu et al., “Interprocedural Exception Analysis for C++”.

[AV19] Artho et al., “Java Pathfinder at SV-COMP 2019 (Competition Contribution)”.

[CKS19] Cordeiro et al., “JBMC: Bounded Model Checking for Java Bytecode”.

[MH20] Mues et al., “JDart: Dynamic Symbolic Execution for Java Bytecode (Competition Contribution)”.

Table 4.3: Comparison of the new exception code with a DIVINE-3-style version.

category	# mod	time (D4)	time (D3)	# states (D4)	# states (D3)
alg	9	3:52	3:51	543.3 k	543.3 k
pv264	13	1:34	1:32	183.0 k	183.0 k
iv112	11	25:58	25:57	3743 k	3743 k
libcxx	425	42:15	42:09	2182 k	2182 k
bricks	292	3:04:25	2:56:55	6271 k	6251 k
divine	3	6:20	6:18	1040 k	1040 k
cryptopals	3	0:01	0:01	1943	1943
llvm	12	36:36	36:27	3865 k	3865 k
svc-pthread	17	16:47	16:41	1685 k	1685 k
total	794	5:21:44	5:13:49	20.1 M	20.0 M

support coverage), “bricks” contains unit tests for various C++ helper classes, including concurrent data structures, “divine” contains unit tests for a concurrent hashset implementation used in DIVINE, “cryptopals” contains solutions of the cryptopals problem set²⁶, the “llvm” category contains programs from the LLVM test-suite²⁷ and finally, the “svc-pthread” category includes pthread-based C programs from the SV-COMP benchmark set. In most of the programs, it was assumed that `malloc` and `new` never fail, with the notable exception of part of the “bricks” category unit tests. The tests where `new` failures are allowed are especially suitable for evaluating exception code, in particular if multiple concurrent threads are running at the time of the possible failure.

4.9.2 Comparison to Builtin Exception Support

In addition to the approach presented in this chapter, we have implemented the approach described in [RBB16] in the context of DIVINE 4. This allowed us to directly measure the penalty associated with the present approach, which is more thorough and less labour-intensive at the same time. Our expectation was that this would translate to slower verification, since the off-the-shelf code is more complex than the corresponding hand-tailored version used in [RBB16]. In line with this expectation, we set the criterion of viability: we would consider a slowdown of at most 10 % to be an acceptable price for the improved verification fidelity, and convenience of implementation. Since other resource consumption (especially memory) of verification is typically proportional to state space size, we have used the number of states explored as an additional metric. The expected effect on the shape (and, by extension, size) of the state space should be smaller than the effect on computation time (most of the additional complexity is related to computing a single transition). We believe that an acceptable penalty in this metric would be about 2 % increase.

As can be seen in Table 4.3, the time penalty on our chosen model set is very acceptable – just shy of 2.6 % – and the state space size is

²⁶ <http://cryptopals.com>

²⁷ <http://llvm.org/svn/llvm-project/test-suite/trunk/SingleSource/Benchmarks/Shootout>

[RBB16] Ročkai et al., “Model Checking C++ Programs with Exceptions”.

Table 4.4: Comparison of the new exception code against stubbed exceptions. Compared to Table 4.3, in this case 133 models failed due to the stubs and were excluded. State counts are identical for all models.

category	# mod	time (D4)	time (stub)	# states
alg	9	3:52	3:52	543.3 k
pv264	13	1:34	1:34	183.0 k
iv112	11	25:58	26:00	3743 k
libcxx	392	41:56	41:54	2179 k
bricks	192	35:30	35:21	2378 k
divine	3	6:20	6:19	1040 k
cryptopals	3	0:01	0:01	1943
llvm	12	36:36	36:28	3865 k
svc-pthread	17	16:47	16:43	1685 k
total	661	2:52:30	2:52:08	16.2 M

within 1 % of the older approach [RBB16]. We believe that this small penalty is well justified by the superior verification properties of the new approach.

[RBB16] Ročkai et al., “Model Checking C++ Programs with Exceptions”.

4.9.3 Comparison to Stub Exceptions

The second alternative approach is to consider any thrown exception an error, regardless of whether it is caught or not. This can be achieved much more easily than real support for exceptions, since we can simply replace the entire `libunwind` interface with stubs which raise an error and refuse to continue. This approach only works for models which do not actually throw any exceptions during their execution. The results of this comparison are shown in Table 4.4 – the verification time is nearly identical and the state spaces are entirely so. This is in line with expectations: in those models, catch blocks are present but never executed. Since the proposed approach does not incur any overhead until an exception is actually thrown, we would not expect a substantial time difference.

4.9.4 Comparison to No Exceptions

Finally, the last alternative is to disable exception support in the C++ frontend entirely. In `clang`, this is achieved by compiling the source code with the `-fno-exceptions` flag. In this case, the LLVM bitcode contains no exception-related artefacts at all, but many programs fail to build. Additionally, a number of programs in the “bricks” category contain exception handlers for memory allocation errors and therefore exit cleanly upon memory exhaustion. Even though some of those programs can be compiled with `-fno-exceptions`, they now contain an error (a null pointer dereference) which is not present when they are compiled the standard way.²⁸ Those programs were therefore excluded from the comparison. The summary of this comparison can be found in Table 4.5 – the time saved for models where `-fno-exceptions` is applicable is again quite small, less than 13 %. In this case, the difference

²⁸ In this case, the handler is installed using `std::set_terminate`, which is available even when `-fno-exceptions` is given. The situation would be similar if only parts of the program were compiled with `-fno-exceptions`. In particular, the problem is that the standard library, if compiled with `-fno-exceptions`, cannot throw, and must therefore behave differently in those scenarios, affecting the behaviour of the user program.

Table 4.5: Comparison of the new exception support against a case where `-fno-exceptions` was used to compile the sources and libraries. In this case, it was only possible to verify 423 models from the set (i.e. 371 models are missing from the comparison). State counts are identical for all models.

category	# mod	time (D4)	time (nxc)	# states
alg	1	0:24	0:23	34.2 k
pv264	1	0:00	0:00	57
iv112	10	23:58	22:06	3571 k
libcxx	393	41:57	40:44	2180 k
svc-pthread	17	16:47	15:42	1685 k
total	423	1:23:33	1:19:21	7504 k

is due to the changes in control flow of the resulting LLVM bitcode. Since `call` is not a terminator instruction (does not perform a jump; unlike `invoke`), the *local* control flow in a function is negatively affected by the presence of `invoke` instructions: more branching is required, and this slows down the evaluator in DiVM. While it is easy to see if a given program can be compiled with `-fno-exceptions`, it is typically much harder to ensure that its behaviour will be unchanged. For this reason, we do not consider the time penalty in verification of this type of programs a problem.

4.9.5 Reusability

As outlined in Section 4.1, the two components directly involved in exception support are comparatively small and well isolated. The LLVM transformation is fully reusable with any LLVM-based tool. The unwinder, on the other hand, relies on the capabilities of DiVM. However, there is no need for hypercalls specific to exception handling and therefore, the implementation work is essentially transparent to DiVM. The capabilities of DiVM required by the unwinder are limited to the following: linked-list stack representation, runtime access to the program frame layout and 2 hypercalls: `__vm_control` and `__vm_obj_free`. More details about DiVM can be found in [RŠČB18].

Finally, adding support for a new type of exceptions is also much simpler in this approach – no modifications to DiVM (or any other host tool) are required: only the two components described in this chapter may need to be modified.

4.10 Conclusion

In this work, we have discussed an approach to extending an LLVM-based model checker with C++ exception support. We have found that reusing an existing implementation of the runtime support library is a viable approach to obtain complete, standards-compliant exception support. A precondition of this approach is that the verification tool is flexible enough to make stack unwinding possible. The DiVM language,

[RŠČB18] Ročkai et al., “DiVM: Model checking with LLVM and graph memory”.

on which the DIVINE model checker is based, has proven to be a good match for this approach, due to its simple and explicit stack representation, along with a suitable set of control flow primitives.

We also performed a survey of tools based on partial or complete reimplementations of C++ exception support routines and found that in each tool, at least one edge case is not well supported. In contrast to this finding, with our approach, all those edge cases are covered “for free”, that is, by the virtue of reusing an existing, complete implementation. Contrary to the prediction made in [RBB16], we have found that with a suitable target language, implementing a new unwinder can be relatively simple. The unwinder implementation described in this chapter is only about 210 lines of C++ code, while it would be impossible to implement without verifier modifications in DIVINE 3. Therefore, we can conclude that with the advent of the DiVM specification [RŠČB18] and its implementation in DIVINE 4, reimplementing the `libunwind` API and reusing `libc++abi` became a viable strategy to provide exception support.

[RBB16] Ročkai et al.,
“Model Checking C++ Programs with Exceptions”.

[RŠČB18] Ročkai et al.,
“DiVM: Model checking with LLVM and graph memory”.

Chapter 5

Analysis of Programs Under the x86-TSO Relaxed Memory Model

Text of this chapter is based on [ŠB18]. The benchmarks are unmodified from the original paper.

This chapter presents an extension of the DIVINE model checker that allows for analysis of C and C++ programs under the x86-TSO relaxed memory model. We use an approach in which the program to be verified is first transformed so that it encodes the relaxed memory behaviour, and after that, it is verified by an explicit-state model checker supporting only the standard sequentially consistent memory. The novelty of our approach is in careful design of encoding of x86-TSO operations so that the nondeterminism introduced by the relaxed memory simulation is minimised. In particular, we allow for nondeterminism only in connection with memory fences and load operations of those memory addresses that were written to by a preceding store. We evaluate and compare our approach with the state-of-the-art bounded model checker CBMC [CKL04; KT14] and stateless model checker Nidhugg [Abd+15]. For the comparison, we employ SV-COMP concurrency benchmarks that do not exhibit data nondeterminism, and we show that our solution built on top of the explicit-state model checker outperforms both of the other tools. The implementation is publicly available as an open source software.

5.1 Motivation and Introduction

Almost all contemporary processors exhibit *relaxed memory behaviour*, which is caused by cache hierarchies, instruction reordering, and speculative execution. This, together with the rise of parallel programs, means that programmers often have to deal with the added complexity of programming under relaxed memory. The behaviour of relaxed memory can be highly unintuitive even on x86 processors, which have stronger memory model than most other architectures. Therefore, programmers often have to decide whether to stay relatively safe with higher-level synchronisation constructs such as mutexes, or whether to tap to the

[ŠB18] Štill et al., “Model Checking of C++ Programs Under the x86-TSO Memory Model”.

[CKL04] Clarke et al., “A Tool for Checking ANSI-C Programs”.

[KT14] Kroening et al., “CBMC – C Bounded Model Checker”.

[Abd+15] Abdulla et al., “Stateless Model Checking for TSO and PSO”.

full power of the architecture and risk subtle unintuitive behaviour of relaxed memory accesses. For these reasons, it is highly desirable to have robust tools for finding bugs in programs running under relaxed memory.

Our aim is primarily to help with the development of lock-free data structures and algorithms. Instead of using higher-level synchronisation techniques such as mutexes, lock-free programs use low-level atomic operations provided by the hardware or programming language to ensure correct results. This way, lock-free programs can exploit the full power of the architecture they target, but they are also harder to design, as the ordering of operations in the program has to be considered very carefully. We believe that by providing a usable validation procedure for lock-free programs, more programmers will find courage to develop fast and correct programs. Recently, many techniques for analysis and verification which take relaxed memory into account have been developed, and research in this field is still pretty active. In this work, we are adding a new technique which we hope will make the analysis of C and C++ programs targeting x86 processors easier.

[Sew+10] Sewell et al.,
“X86-TSO: A Rigor-
ous and Usable Pro-
grammer’s Model for
x86 Multiprocessors”.

We extend DIVINE with the support for the x86-TSO memory model [Sew+10] which describes the relaxed behaviour of x86 and x86-64 processors. Due to the prevalence of the Intel and AMD processors with the x86-64 architecture, the x86-TSO memory model is a prime target for program analysis. It is also relatively strong and therefore underapproximates most of the other memory models – any error which is observable on x86-TSO is going to manifest itself under the more relaxed POWER or ARM memory models. More details about x86-TSO can be found in Section 2.2.3.

To verify a program under x86-TSO, we first transform it by encoding the semantics of the relaxed memory into the program itself, i.e., the resulting transformed program itself simulates nondeterministically relaxed memory operations. To reveal an error related to the relaxed memory behaviour, it is then enough to verify the transformed program with a regular model checker supporting only the standard sequentially consistent memory.

In this chapter, we introduce a new way of encoding the relaxed memory behaviour into the program. Our new encoding introduces low amount of nondeterminism, which is the key attribute that allows us to tackle model checking of nontrivial programs efficiently. In particular, we achieve this by delaying nondeterministic choices arising from x86-TSO as long as possible. Our approach is based on the standard operational semantic of x86-TSO with store buffers, but it removes entries from the store buffer only when a load or a fence occurs (or if the store buffer is bounded and full). Furthermore, in loads, we only remove those entries from store buffers that relate to the address being loaded, even if there are some older entries in the store buffer.

5.2 Conventional Semantics of x86-TSO

To better illustrate the advantages of our semantics for x86-TSO, we will now present the more usual semantics of this memory model often

used in verification. This semantics is based on the x86-TSO semantics presented in [Sew+10]. It uses a (possibly bounded) thread-local store buffer which stores memory store entries. The oldest entry in the store buffer can be flushed at any point, yielding all possible ways stores can be delayed (up to the reordering bound). Often, the nondeterministic flushing is achieved by a *flusher* thread associated with each store buffer – this allows uniform modelling of thread and memory model nondeterminism. This approach is used for example by [ZKW15] and [Abd+17] or in our previous work in [ŠRB16].

The disadvantage of this semantics is that it introduces a lot of nondeterminism. Once there are any entries in the store buffer, the store buffers can be (partially) flushed at any point the thread rescheduling is possible. Therefore, without an additional reduction, there will often be a lot of runs that differ only in the moment a particular value was flushed from the store buffer to the main memory, even if this value is never read (or is read much later). We illustrate this behaviour in Figure 5.1.

In practice, analysers for parallel programs employ state space reductions which can remove some of the nondeterminism introduced by the flusher thread. However, the flusher thread can behave somewhat differently from normal program threads – it is not necessary to execute the flusher thread at all if the store buffer it belongs to does not contain a value which will be read and it suffices to execute it just before such a read. Taking this into account requires either changes to the reduction mechanism, or to the semantics of x86-TSO simulation. We have decided to take the second route and show that it opens room for further optimisations not possible with the conventional flusher-thread implementation.

5.3 x86-TSO in DIVINE

DIVINE does not natively support relaxed memory, and we decided not to complicate the already complex execution engine and memory representation with a simulation of relaxed behaviour. Instead, we encode the relaxed behaviour into the program itself on the level of LLVM intermediate representation. The modified program running under sequential consistency simulates all x86-TSO runs of the original program, up to some bound on the number of stores which can be delayed. The program transformation is rather similar to the one presented in our previous work in [ŠRB16]. The main novelty is in the way of simulation of x86-TSO which produces significantly less nondeterminism and therefore substantial efficiency improvements.

5.3.1 Simulation of the x86-TSO Memory Model

The most straightforward way of simulating x86-TSO is to add store buffers to the program and flush them nondeterministically, for example using a dedicated flusher thread which flushes one entry at a time and interleaves freely with all other threads. We used this technique in [ŠRB16]. This approach does, however, create many redundant interleavings as the flusher thread can flush an entry at any point,

[Sew+10] Sewell et al., “X86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors”.

[ZKW15] Zhang et al., “Dynamic Partial Order Reduction for Relaxed Memory Models”.

[Abd+17] Abdulla et al., “Stateless model checking for TSO and PSO”.

[ŠRB16] Štill et al., “Weak Memory Models as LLVM-to-LLVM Transformations”.

```

1 void t0() {
2   x = 1;
3   y = 1;
4 }
1 void t1() {
2   z = 1;
3 }
1 void t2() {
2   int a = y;
3   int b = x;
4   assert(!(a == 0 && b == 1));
5 }

```

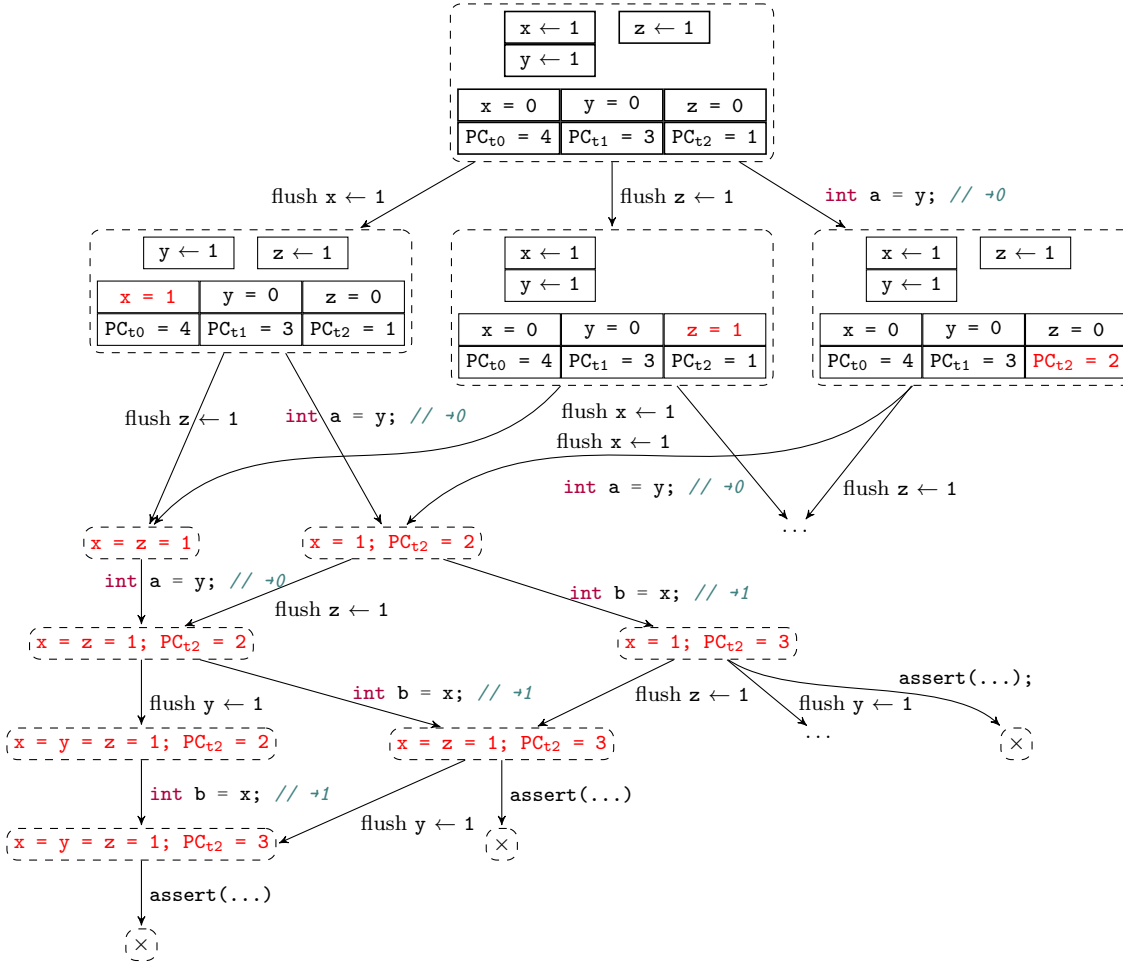


Figure 5.1: An example parallel program and a fragment of its state space. We start in a state where the threads t_0 and t_1 had already executed but did not flush their buffers yet and t_2 did not start executing. This fragment of the state space shows (some of) the states that reach the assertion violation, i.e., states where $a = 0 \wedge b = 1$. The states marked with \times are the states which have reached the assertion violation. States in the first two rows of the state space show store buffers on the top and state of the memory and program counters on the bottom. In the rest of the state space, we show only differences in the memory and program counters.

We can see that the state space contains many redundant runs. First, flushes of $z \leftarrow 1$ are irrelevant as z is never read. The same holds for flushes of $y \leftarrow 1$ after the execution of line 2 in t_2 . Finally, flushes of $x \leftarrow 1$ are only relevant just before the execution of line 2 in t_2 , which is the only one which reads x .

regardless of whether or not it is going to produce a run with a different memory access ordering, i.e. without any respect to whether the flushed value is going to be read or not.

To alleviate this problem, it is possible to delay the choice whether to flush an entry from a store buffer to the point when the first load tries to read a buffered address. Only when such a load is detected, all possible ways the store buffers could have been flushed are simulated. In this case, the load can trigger flushing from any of the store buffers, to simulate that they could have been flushed before the load. To further improve the performance, only entries relevant to the loaded address are affected by the flushing. These are the entries with matching addresses and any entries which precede them in the corresponding store buffers (that are flushed before them to maintain the store order).

A disadvantage of this approach is that there are too many ways in which a store buffer entry can be flushed, especially if this entry is not the oldest in its store buffer, or if there are entries concerning the same addresses in multiple store buffers. All of these cases can cause many entries to be flushed, often with a multitude of interleavings of entries from different store buffers which has to be simulated.

Delayed Flushing To alleviate aforementioned explosion of possible orders in which the store buffers can be flushed, we propose *delayed flushing*: entries in the store buffers can be kept in the store buffer after newer entries were flushed if the retained entries are marked as *flushed*. Such the entries behave as if they were already written to the main memory, but can still be reordered with entries in other store buffers. That is, when there is a flushed entry for a given location in any store buffer, the value stored in the memory is irrelevant as any load will either read the flushed entry or entry from the other store buffer (which can be written after the flushed entry). Flushed entries make it possible to remove store buffer entries out of order while preserving total store order of observable operations. This way, a load only affects entries from the matching addresses and not their predecessors in the store order. This improvement is demonstrated in Figure 5.2.

Program Transformation DIVINE handles C and C++ code by translating it to LLVM and instrumenting it (see Figure ?? for DIVINE's workflow). The support for relaxed memory is added in the instrumentation step, by replacing memory operations with calls to functions which simulate relaxed behaviour. Essentially, all loads, stores, atomic compound instructions¹, and fences are replaced by calls to the appropriate functions.² All of the x86-TSO-simulating functions are implemented so that they are executed atomically by DIVINE (i.e., not interleaved).

x86-TSO Memory Operations The most complex of these is the load operation. It first finds all entries with overlap the loaded address (*matching entries*) and out of these matching entries, it nondeterministically selects entries which will be written before the load (*selected entries*). All matching entries marked as flushed have to be selected for writing. Similarly, all matching entries which occur in a store buffer

¹ Compare and swap, atomic exchange, and compound arithmetic and logic operations such as fetch and add

² With the exception of atomic compound operations, all modifications to the memory in LLVM must be performed by a series of instructions containing a load to a register, modifications of the register, and store of register value to the memory. Therefore it is sufficient to consider only this limited number of instructions as other instructions do not access memory at all.

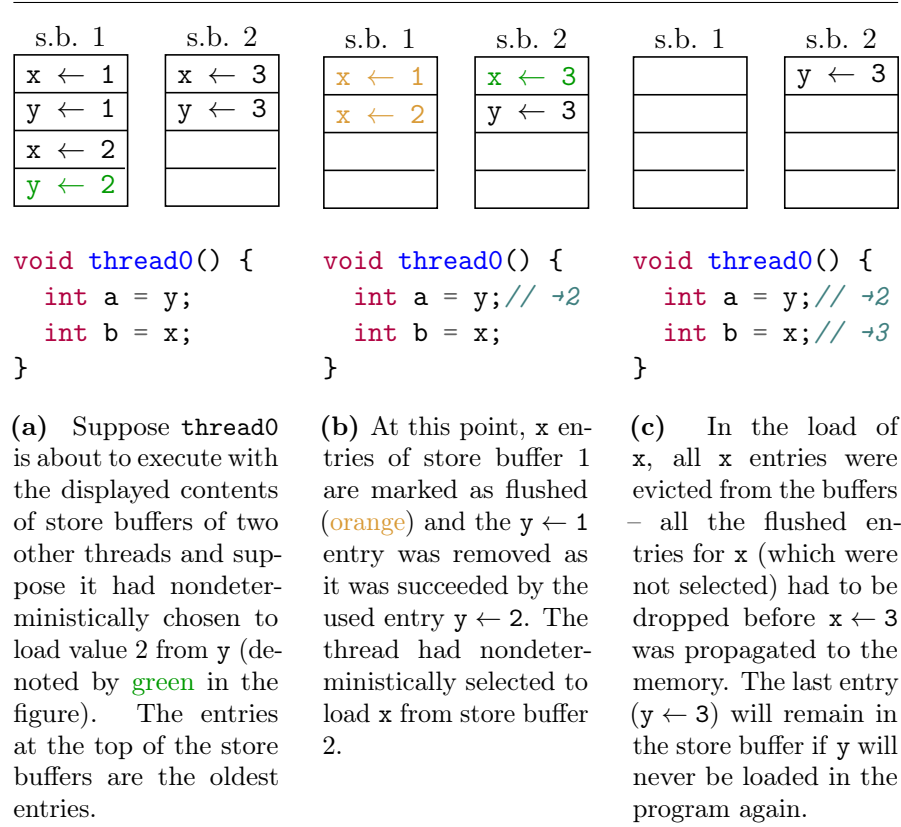


Figure 5.2: An illustration of the *delayed flushing*.

before a selected entry also have to be selected. Out of the selected entries, one is selected to be written last – this will be the entry read by the load. Next, selected entries are written, and all nonmatching entries which precede them are marked as flushed. Finally, the load is performed, either from the local store buffer if matching entry exists there, or from the shared memory.

The remaining functions are relatively straightforward. Stores push a new entry to the store buffer, possibly evicting the oldest entry from the store buffer if the store buffer exceeds its size bound. Fences flush all entries from the store buffer of the calling thread to the memory. Atomic operations are basically a combination of a load, store, and a fence – all atomics are sequentially consistent under x86-TSO. The only intricate part of these operations is that if an entry is flushed out of the store buffer, the entries from other store buffers which involve the same memory location can also be nondeterministically flushed (to simulate they could have been flushed before the given entry). This flushing is similar to flushing performed in load. An example which shows a series of loads can be found in Figure 5.2.

Correctness We will now argue that this way of implementing x86-TSO is correct. First, the nondeterminism in selecting entries to be flushed before a load serves the same purpose as the nondeterminism in the flusher thread of the more conventional implementation. The only difference is that in the flusher-thread scenario the entries

are flushed in order, while in our new approach we are selecting only from the matching entries. Therefore, the difference between the two approaches is only on those entries which are not loaded by the load causing the flush, hence cannot be observed by the load. However, any entry which would be flushed before the selected entries in the flusher-thread approach is now marked with the flushed flag. This flag makes sure that such an entry will be flushed before an address which matches it is loaded, and therefore it behaves as if it was flushed. This way, the in-thread store order is maintained.

5.3.2 Stores to Freed Memory

As x86-TSO simulation can delay memory stores, special care must be taken to preserve memory safety of the program. More precisely, it is necessary to prevent the transformed program from writing into freed memory. This problem occurs if a store to dynamically allocated memory is delayed after the memory is freed, or if a store to stack location is delayed after the corresponding function had returned. This problem does not require special handling in normal program execution as both stack addresses as well as dynamic memory addresses remain to be writable for the program even after they are freed (except for memory-mapped files, but these have to be released by a system call which includes sufficiently strong memory barrier which prevents memory accesses from being delayed past the system call).

To solve the problem of freed memory, it is necessary to evict store buffer entries which correspond to the freed memory just before the memory is freed. For entries not marked as flushed, this eviction concerns only store buffer of the thread which freed the memory. If some other thread attempted to write to the freed memory, this is an error as there is insufficient synchronisation between the freeing and the store to the memory. However, corresponding entries marked as flushed should be evicted from all store buffers, as these entries correspond to changes which should have been already written to the shared memory.

Eviction of dynamic memory is straightforward – the program transformation injects a call to the eviction function just before every call to the function which releases memory. For eviction of stack memory, it is necessary to evict all local addresses whenever a function exits, regardless of the way it exits. This means we also have to take into account exceptions and other ways of performing non-local transfers of control (e.g., `longjmp`). The program transformation takes care of tracking which local memory addresses should be evicted and inserts code to evict them at the end of functions.

5.3.3 Integration with Other Parts of DIVINE

The integration of x86-TSO simulation with the rest of DIVINE is straightforward in most cases. No changes are required in the DIVINE's execution engine or state space exploration algorithms. As for the libraries shipped with DIVINE, only minor tweaks were required. The `pthread` implementation had to be modified to add full memory barrier both at the beginning and at the end of every synchronising functions.

This corresponds to barriers present in the implementations used for normal execution, `pthread` mutexes and other primitives have to guarantee sequential consistency of the guarded operations (provided all accesses are properly guarded).

[Roč+19] Ročkai et al., “Reproducible Execution of POSIX Programs with DiOS”.

The DIVINE’s operating system, DiOS, is used to implement low-level threading as well as simulation of various filesystem APIs [Roč+19]. We had to add memory barrier into the system call entry which hands control to DiOS. DiOS itself does not use relaxed memory simulation – the implementation of x86-TSO operations detects that the code is executed in the kernel mode and bypasses store buffers. In this way, the entire DiOS executes as if under sequential consistency which makes its design simpler and more robust. This synchronisation is easily justifiable – system calls require a memory barrier or kernel lock in most operating systems.

[LRB18] Lauko et al., “Symbolic Computation via Program Transformation”.

Relaxed Memory and Abstract Data DIVINE has support for symbolic and abstract data (data nondeterminism) which uses program transformation to implement symbolic manipulation of data [LRB18]. However, the integration of support for abstract data and for memory models is not yet done as it requires more complex changes than just a straightforward composition of the two program transformations. Performing transformation for abstractions first is not possible currently, as it introduces additional memory operations and therefore is not correct in presence of relaxed memory. On the other hand, it should be possible to run the transformation which introduces relaxed memory behaviour first, and only after that run the transformation which introduces abstraction. However, this would mean that if an abstract value is ever stored to memory (i.e., it is not only present in LLVM registers) then all values loaded from the memory are potentially abstract, because the store buffers themselves need to be abstract. Therefore, this approach is not practical because it introduces too much extra overhead. Overall, we defer this problem to future work.

5.3.4 Further Optimizations

We have implemented two further optimisations of our x86-TSO simulation.

Static Local Variable Detection Accesses of local variables which are not accessible to other threads need not use store buffering. For this reason, we have inserted a static analysis pass which annotates accesses to local memory before the x86-TSO instrumentation. The instrumentation ignores such annotated accesses. The static analysis can detect most local variables which are never accessed using pointers.

Dynamic Local Memory Detection DIVINE can also dynamically detect if the given memory object is shared between threads (i.e., it is accessible from global variables or stacks of more than one thread). Using this information, it is possible to dynamically bypass store buffers for operations with non-shared memory objects. This optimisation is correct even though the shared status of memory can change during its

lifetime. A memory object o can become shared only when its address is written to some memory object s which is already shared (or o can become shared transitively through a series of pointers and intermediate objects). For this to happen, there has to be a store to the already shared object s , and this store has to be propagated to other threads. Once the store of the address of o is executed and written to the store buffer, o becomes shared, and any newer stores into it will go through the store buffer. Furthermore, once this store is propagated, any store which happened before turning o into a shared object also had to be propagated as x86-TSO does not reorder stores. Therefore, there is no reason to put stores to o through the store buffer if o is not shared. This optimisation is not correct for memory models which allow store reordering – for such memory models, we would need to know that the object will never be shared during its lifetime.

5.3.5 Bounding the Size of Store Buffers

The complexity of analysis of programs under the x86-TSO memory model is high. From the theoretical point of view, we know due to [ABBM10] that reachability for programs with finite-state threads which run under TSO is decidable, but non-primitive recursive (it is in PSPACE for sequential consistency). The proof uses the so-called SPARC TSO memory model [SPA94] that is very similar to x86-TSO. However, the proof of decidability does not translate well to an efficient decision procedure, and real-world programs are much more complex than the finite-state systems used in the decidability proof.

For this reason, we would need to introduce unbounded store buffers to properly verify real-world programs. Unfortunately, this can be impractical, especially for programs which do not terminate. Therefore, our program transformation inserts store buffers of limited size, limiting thus the number of store operations that can be delayed at any given time. The size of the store buffers is fully configurable, and it currently defaults to 32, a value probably high enough to discover most bugs which can be observed on real hardware.

Our implementation also supports the store buffers of unlimited size (when size is set to 0). In this mode, programs with infinite loops that write into shared memory will not have finite state space under x86-TSO even if they have finite state space under sequential consistency. Therefore, DIVINE will not terminate unless it discovers an error in the program. Verification with unbounded buffers will still terminate for terminating programs and for all programs with errors.

5.4 Evaluation

The implementation is available at <http://divine.fi.muni.cz/2018/x86tso/>, together with information about how to use it. It is also integrated into DIVINE releases. We compared our implementation with the stateless model checker Nidhugg [Abd+15] and the bounded model checker CBMC [CKL04; KT14]. For evaluation we used SV-COMP 2017 benchmarks from the Concurrency category [Bey17], excluding benchmarks with data nondeterminism³ as DIVINE does not yet support

[ABBM10] Atig et al., “On the Verification Problem for Weak Memory Models”.

[SPA94] SPARC International, “The SPARC Architecture Manual”.

[Abd+15] Abdulla et al., “Stateless Model Checking for TSO and PSO”.

[CKL04] Clarke et al., “A Tool for Checking ANSI-C Programs”.

[KT14] Kroening et al., “CBMC – C Bounded Model Checker”.

[Bey17] Beyer, “Software Verification with Validation of Results”.

³ I.e., all the benchmarks which contain calls to functions of the `__VERIFIER_nondet_*` family were excluded.

Table 5.1: Comparison of the default configuration of DIVINE with CBMC and Nidhugg.

	CBMC	Nidhugg	DIVINE
finished	21	25	27
TSO bugs	3	3	9
unique	5	3	5

data nondeterminism with relaxed memory. Furthermore, due to the limitation of stateless model checking with DPOR, Nidhugg cannot handle data nondeterminism at all. There are 55 benchmarks in total.

The evaluation was performed on a machine with two dual core Intel Xeon 5130 processors running at 2 GHz with 16 GB of RAM. Each tool was running with memory limit set to 10 GB and time limit set to 1 hour. The tools were not limited in the number of CPUs they can use.

We have used CBMC version 5.8 with the option `--mm tso`. Since there is no official release of Nidhugg, we have used version 0.2 from git, commit id 375c554 with `-tso` option to enable relaxed memory support and inserted a definition of the `__VERIFIER_error` function. For DIVINE, we have used the `--svcomp` option to enable support for SV-COMP atomic sections (which are supported by default by CBMC and Nidhugg), and we disabled nondeterministic memory failure by using the `divine check` command (SV-COMP does not consider the possibility of allocation failure). To enable x86-TSO analysis, `--relaxed-memory tso` is used for DIVINE.⁴ The buffer bound was the default value (32) unless stated otherwise.

Table 5.1 compares performance of the default configuration of DIVINE with the remaining tools. The line “finished” shows the total number of benchmarks for which the verification task finished with the given limits. From these the line “TSO bugs” shows the number of errors caused by relaxed memory in benchmarks which were not supposed to contain any bugs under sequential consistency. All discovered errors were manually checked to really be observable under the x86-TSO memory model. Finally, “unique” shows the number of benchmarks solved only by the given tool and not the other two. There were only 8 benchmarks solved by all three tools, all of them without any errors found.

Table 5.2 shows effects of buffer size bound and improvements described in Section 5.3.4. It can be seen that all versions perform very similarly, only one more benchmark was solved by the versions with dynamic shared object detection (the remaining solved benchmarks were the same for all versions). The number of solved benchmarks remains the same regardless of used store buffer bound.

Table 5.3 offers more detailed look at the 26 benchmarks solved by all versions of DIVINE. It shows the aggregate differences in state space sizes and solving times. It can be seen that the dynamic shared object detection improves performance significantly. Interestingly, we can see that of the 3 versions which differ only in store buffer size

⁴ The complete invocation is `divine check --svcomp --relaxed-memory tso BENCH.c`.

Table 5.2: Comparison of various configurations of DIVINE. The “base” version uses none of the improvements from Section 5.3.4. The configurations marked with “s” add the static local variable optimisation, while the configurations marked with “d” add the dynamic detection of non-shared memory objects. The “+sdu” configuration has both optimisations enabled and it has unbounded buffers. Finally, the “+sd4” has buffer bound set to 4 entries instead of the default 32 entries. The default version is “+sd”.

	base	+s	+d	+sd	+sdu	+sd4
finished	26	26	27	27	27	27
TSO bugs	8	8	9	9	9	9

Table 5.3: Comparison of various versions of DIVINE on benchmarks on the 26 which all the versions finished. For the description of these versions, refer to Table 5.2.

	base	+s	+d	+sd	+sdu	+sd4
states	252 k	263 k	250 k	231 k	206 k	296 k
time	2:14:49	2:17:13	1:09:23	1:05:05	0:58:28	1:24:59

(“+sd”, “+sdu”, and “+sd4”), the unbounded version performs the best. We expect this to be caused by the nondeterminism in flushing the excessive entries out of the store buffer when the bound is reached – this can trigger flushing of matching entries from other store buffers and therefore increase nondeterminism.

5.5 Related work

A lot of techniques have support for program analysis under relaxed memory models, most of these were already described in Chapter 3 and therefore, we will not replicate the description here. A lot of these techniques is build on stateless model checking (see also Section 3.2) including [DL15; ZKW15; HH16; Abd+17] for x86-TSO and [AAJL16; ND16; KLSV17; AAJN18] for other memory models. Bounded model checking (see also Section 3.4) is another popular choice for analysis of x86-TSO [AKT13; Gav+19; Tom+17] and other memory models [BAM07; AABN17]. An explicit-state approach to programs running under the C# relaxed memory model is presented in [HH16]. There are also previous works which use program transformation in the analysis of relaxed memory models [AKNT13; AABN17; ŠRB16].

Our approach to simulation of x86-TSO is somewhat similar to the approach presented in [AAJN18]. Both approaches perform nondeterministic choice primarily on memory loads. However, [AAJN18] is focusing on the release-acquire fragment of C11 and does not support sequential consistency. Similarly, [KLSV17] presents a technique that attempts to minimise nondeterminism, this time in the analysis of a modified version of the C11 memory model.

[DL15] Demsky et al., “SATCheck: SAT-Directed Stateless Model Checking for SC and TSO”.

[ZKW15] Zhang et al., “Dynamic Partial Order Reduction for Relaxed Memory Models”.

[HH16] Huang et al., “Maximal Causality Reduction for TSO and PSO”.

[Abd+17] Abdulla et al., “Stateless model checking for TSO and PSO”.

[AAJL16] Abdulla et al., “Stateless Model Checking for POWER”.

[ND16] Norris et al., “A Practical Approach for Model Checking C/C++11 Code”.

[KLSV17] Kokologiannakis et al., “Effective Stateless Model Checking for C/C++ Concurrency”.

[AAJN18] Abdulla et al., “Optimal Stateless Model Checking under the Release-Acquire Semantics”.

[AKT13] Alglave et al., “Partial Orders for Efficient Bounded Model Checking of Concurrent Software”.

[Gav+19] Gavrilenko et al., “BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings”.

[Tom+17] Tomasco et al., “Using Shared Memory Abstractions to Design Eager Sequentializations for Weak Memory Models”.

[BAM07] Burckhardt et al., “CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models”.

[AABN17] Abdulla et al., “Context-Bounded Analysis for POWER”.

[AKNT13] Alglave et al., “Software Verification for Weak Memory via Program Transformation”.

[ŠRB16] Štill et al., “Weak Memory Models as LLVM-to-LLVM Transformations”.

[ABP11] Atig et al., “Getting Rid of Store-Buffers in TSO Analysis”.

[LW10] Linden et al., “An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models”.

[Hol97] Holzmann, “The model checker SPIN”.

[BCDM15] Bouajjani et al., “Lazy TSO Reachability”.

[DMVY13] Dan et al., “Predicate Abstraction for Relaxed Memory Models”.

[BM08] Burckhardt et al., “Effective Program Verification for Relaxed Memory Models”.

[BDM13] Bouajjani et al., “Checking and Enforcing Robustness against TSO”.

There are also some approaches to the analysis of relaxed memory models which were not mentioned in Chapter 3, mostly because they lack implementation which works with realistic programming languages. In [ABP11] a program transformation that uses context switch bounding but not buffer bounding is presented. It uses additional copies for shared variables for TSO simulation. However, the evaluation was performed with manually converted programs. An unbounded approach to verification of programs under TSO is presented in [LW10]. It uses store buffers represented by automata and leverages cycle iteration acceleration to get a representation of store buffers on paths which would form cycles if values in store buffers were disregarded. It targets a modified Promela modelling language [Hol97]. Another unbounded approach is presented in [BCDM15]. It introduces TSO behaviours lazily by iterative refinement, and while it is not complete, it should eventually find all errors. The work [DMVY13] presents verification of (potentially infinite state space) programs under TSO and PSO (with bounded store buffers) using predicate abstraction. It first analyses the program under sequential consistency and then extrapolates the predicates to TSO/PSO and performs further analysis (and possibly refinement).

Absence of Relaxed Behaviour There are also techniques which aim to solve the problem of absence of relaxed behaviour in a given program. For these methods, the question is whether a program, when running under a relaxed memory model, exhibits any runs not possible under sequential consistency. This problem is explored under many names, e.g. (TSO-)safety [BM08], robustness [BDM13; DM14], stability [AM11], and monitoring of sequential consistency [BSS11]. Similar techniques are used in [YGL04] to detect data races in Java programs. A related problem of correspondence between a parallel and sequential implementation of a data structure is explored in [OD17]. Some of these techniques can be also used to insert memory fences into the programs to recover sequential consistency.

Neither of these techniques are directly comparable to our method. For these techniques, a program is incorrect if it exhibits relaxed behaviour, while for us, it is incorrect if it violates specification (e.g., assertion safety and memory safety). In practice, the appearance of relaxed behaviour is often not a problem, provided the overall behaviour of the data structure or algorithm matches the desired specification. In many lock-free data structures, relaxed behaviour is essential to achieving high performance.

Other Methods In [PD95], the SPARC hierarchy of memory models (TSO, PSO, RMO) is modelled using encoding from assembly to $\text{Mur}\phi$ [Dil96]. A completely different approach is taken in [TVD14]. This work introduces a separation logic GPS, which allows proving properties about programs using (a fragment of) the C11 memory model. This work is intended for manual proving of properties of parallel programs, not for automatic verification. The memory model used in this work is not complete; it lacks relaxed and release-consume accesses.

Another fragment of the C11 memory model is targeted by the RSL separation logic introduced in [VN13].

5.6 Conclusion

We showed that by careful design of simulation of relaxed memory behaviour we can use the standard model checker supporting only the sequential consistency to efficiently detect relaxed memory errors in programs that are otherwise correct under sequentially consistent memory. Moreover, according to our experimental evaluation, our explicit-state model checking approach outperforms a state-of-the-art stateless model checker as well as bounded model checker, which is actually quite an unexpected result. We also show that many of the used benchmarks can be solved only by one or two of the three evaluated tools, which highlights the importance of employing different approaches to the analysis of programs under relaxed memory. Finally, we show that for terminating programs, our approach is viable both with bounded and unbounded store buffer size.

[DM14] Derevenetc et al., “Robustness against Power is PSpace-complete”.

[AM11] Alglave et al., “Stability in Weak Memory Models”.

[BSS11] Burnim et al., “Sound and Complete Monitoring of Sequential Consistency for Relaxed Memory Models”.

[YGL04] Yang et al., “Memory-Model-Sensitive Data Race Analysis”.

[OD17] Ou et al., “Checking Concurrent Data Structures Under the C/C++11 Memory Model”.

[PD95] Park et al., “An Executable Specification, Analyzer and Verifier for RMO (Relaxed Memory Order)”.

[Dil96] Dill, “The Murphi Verification System”.

[TVD14] Turon et al., “GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation”.

[VN13] Vafeiadis et al., “Relaxed Separation Logic: A Program Logic for C11 Concurrency”.

Chapter 6

Local Nontermination analysis for Parallel Programs

Text of this chapter is based on [ŠB19]. The benchmarks are unmodified from the original paper.

[ŠB19] Štill et al., “Local Nontermination Detection for Parallel C++ Programs”.

One of the critical problems with parallel programs is ensuring that they do not hang or wait indefinitely – i.e., there are no deadlocks, livelocks and the program proceeds towards its goals. In this work, we present a practical approach to detection of nonterminating sections of programs written in C or C++ and its implementation into the DIVINE model checker. This complements the existing techniques for finding safety violations such as assertion failures and memory errors. Our approach makes it possible to detect partial deadlocks and livelocks, i.e., those situations in which some of the threads are progressing normally while the others are waiting indefinitely. The approach is also applicable to proving nontermination of components in programs that do not terminate themselves, but the components should eventually finish their work. Such programs include, for example, server-like applications that have infinite event loops, but each event should be handled in a finite time. The termination criteria can be user-provided; however, DIVINE comes with the set of built-in termination criteria suited for the analysis of programs with mutexes and other common synchronisation primitives.

6.1 Motivation and Introduction

A significant limitation of many existing tools for analysis of parallel programs in programming languages such as C and C++ is that they are only concerned with safety checking – they check that a bad state of the program is unreachable. Most common examples of bad states include assertion failures and memory errors (such as invalid memory accesses and memory leaks). Unfortunately, this is far from being sufficient in practice. See, for example, the code given in Figure 6.1. That piece of code easily passes any safety checks; however, when executed in reality, it often hangs and does not terminate.

```

// can be used for
// synchronisation
std::atomic< int > x = 0;

void worker() {
    while ( x != 0 ) { } //
    ↪ wait
    do_work();
}

int main() {
    // start thread
    // running worker
    std::thread t( worker );
    x = 42; // let worker run
    // ...
    t.join();
}

```

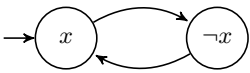
Figure 6.1: A simple C++ code snippet with two threads, it uses C++ standard threading support and atomic variables. A programmer’s intention was that the `worker` function first waits until `x` becomes non-zero, and then proceeds with `do_work`. However, the waiting condition (at the first line of the `worker` function) is incorrectly just the opposite. Therefore, if `main` executes `x = 42` before waiting in `worker` starts, the wait will never end (assuming `x` is never set to 0 again). Note that none of safety checks is able to detect that the program might hang. For the rest of the paper, we will omit the `std::` namespace to simplify the notation.

In this work, we report about our new technique for checking nontermination for parallel programs written in C and C++ that may be applied to programs with arbitrary synchronisation primitives. In particular, we can check that *a specified part* of a program finishes whenever its execution has been started, which in turn enables us to check for problems such as partial deadlocks or local nontermination. Note that our technique does not require the program under analysis to terminate at all. Therefore, it is also applicable to programs that do not terminate but have some parts that are supposed to finish. It does; however, require that the program has a finite state space because our technique is built on top of a state space exploration. Note that even for a finite state space, a program may exhibit infinite behaviour.¹

The main observation is that a program often has sections which once entered should also be left: for example critical sections, certain function calls (such as pop from a queue, which can wait for an element to become available; or a thread join, etc.), or parts of code which wait for a resource or an action (waiting for a mutex, waiting on a barrier, waiting until a variable is set to a given value). If the analysis of the program focuses on such sections, it is possible to detect when these sections are started but do not terminate. This covers partial deadlock and partial livelock detection in which such sections participate. We also provide a global nontermination detection mode that decides if the program as a whole terminates, nevertheless this is not the primary goal of our approach.

Our technique is built on top of explicit-state model checking. We believe that while explicit-state model checking is prone to state space explosion, it is well suited for the detection of problems related to infinite runs of parallel programs which cannot be handled by techniques such as bounded model checking or stateless model checking. While our

¹ A finite state space can contain cyclical infinite behaviour – a loop in the state space.



approach is closely related to checking for properties written in temporal logic such as LTL or CTL*, our *local nontermination* technique cannot be substituted equivalently with CTL* model checking. One of the reasons is that these logics are unable to relate to entities which are dynamically created during the execution of the program, and there is no bound to their number. For example, there is no way to express in CTL* that for all mutexes it holds that if they are locked, they are also eventually unlocked unless all the mutexes are enumerated beforehand. This is an essential concern for realistic programs where mutexes and other synchronisation primitives can be created dynamically at runtime, and their number can depend on the computation of the program itself. Furthermore, to avoid counterexamples which are unrealistic with practical thread schedulers, we need a form of *fairness* of process scheduling different from the fairness constraints used typically with LTL model checking.

The approach described in this work is implemented in a modified version of the DIVINE model checker [Bar+17; RŠČB18]. The implementation, as well as all the examples, can be found on the paper webpage².

6.2 Resource Sections

A *resource section* of a program is a block of code with an identifier of a resource and type of the resource section. Each resource section is delimited in the source code by section start and section end annotations. Examples of such sections are a mutex-waiting section that denotes a block of code in which a thread is waiting for the acquisition of a mutex. Mutex-waiting section is identified by a mutex and the thread which waits for it. Another example can be a critical section, which is identified by a mutex (there is no need to use a thread for the identification, as a mutex can be owned by at most one thread at any point in time). Resource section can also be bound to a function – in this case, it is identified by the stack frame of the function and by the program counter of its beginning. Regardless of the identification, the idea for a resource section is that once it is entered, it should also be exited.

As a resource section can be entered repeatedly (for example when it is on a cycle or in a function which is called multiple times), we will define a *resource section instance* to be a particular execution of a resource section with the given identifier. The author of annotations which define resource sections should ensure that the same resource section is not entered again before it is left. This does not limit the usage of function-associated resource sections to non-recursive functions – each such section is also identified by the stack frame, and therefore resource sections corresponding to different recursion depths are different resource sections. Similarly, a program can be in multiple resource sections which wait for the same mutex at the same time, each of them corresponding to a different waiting thread.

We propose to introduce resource sections in two ways – some synchronisation primitives (mutexes, condition variable, threads) have

[Bar+17] Baranová et al., “Model Checking of C and C++ with DIVINE 4”.

[RŠČB18] Ročkai et al., “DiVM: Model checking with LLVM and graph memory”.

²<https://divine.fi.muni.cz/2019/Interm/>

```

1  mutex m;
2
3  void worker() {
4      unique_lock lock(m); // wait resource section,
5                          // start of critical section
6      do_work();
7  } // unlock = end of critical section
8
9  int main() {
10     thread w(worker);
11     do_work();
12     w.join(); // wait resource section
13 }

```

Figure 6.2: Examples of predefined resource sections. There is a resource section in the mutex lock on line 4 (active while the thread waits for the mutex to become available), a resource sections which corresponds to the critical section on lines 4–6, and a resource section in the join of thread `w` on line 12 (active while main waits for `worker` to finish).

built-in resource sections in DIVINE, and the user can introduce their own resource sections. Examples of resource sections can be seen in Figure 6.2 and Figure 6.3.

6.3 Local Nontermination

With our local nontermination property, we aim at detection of resource section instances which are entered but are never left – *nonterminating resource section instances*. We will first use examples of terminating and nonterminating resource section instances, and then we will define them precisely.

A simple example can be seen in Figure 6.4a. There we have a mutex which is locked, but never unlocked as the corresponding critical section contains an infinite loop. We have four different resource sections in this example. Two of them corresponds to the critical sections guarded by the mutex, and two of them are hidden inside `unique_lock`, where they implement waiting until the mutex is unlocked. Nonterminating resource section instances are the instances corresponding to the critical section in `thread0` and any instances corresponding to waiting for the mutex in `thread1` that is executed after the critical section in `thread0` is entered. We can fix this example by putting the critical section in `thread0` inside the infinite loop, as shown in Figure 6.4b.

Suppose that we have defined nonterminating section as one in which it is possible to stay indefinitely (i.e., for the specific case of waiting for `m` in `thread1`, termination could be expressed by LTL formula $\mathbf{G}(\text{wait-}m\text{-}t1\text{-start} \implies \mathbf{F} \text{wait-}m\text{-}t1\text{-end})$). We can witness the existence of such nonterminating section in a program with a finite state space by a lasso-shaped path. Such the nontermination witness can also be found for the program in Figure 6.4b, even though the

```

1  #include <atomic>
2  #include <rst/termsec.h> // for user-defined
3                               // resource sections
4  struct SpinLock {
5      void lock() {
6          { // define an explicit scope
7              termsec::CheckWait check( &_flag );
8              while ( _flag.test_and_set() ) { }
9          } // end of scope -- end of 'check' resource section
10         termsec_begin_exclusive( &_flag );
11     }
12
13     void unlock() {
14         termsec_end_exclusive( &_flag );
15         _flag.clear();
16     }
17
18     private:
19         std::atomic_flag _flag = ATOMIC_FLAG_INIT;
20 };

```

Figure 6.3: User defined resource sections in a spinlock implementation. The spinlock is implemented using a C++ atomic flag which is an atomic variable which has two operations – `clear` which resets its value to `false`, and `test_and_set` which sets its value to `true` and returns the original value. In this spinlock the value `true` indicates that the spinlock is locked.

One resource section guards the wait for `_flag` to change at lines 7–8. The other resource section guards the critical section of the spinlock, it starts in `lock` at line 10 and ends in `unlock` at line 14. Both resource sections are identified by the address of the `_flag` variable which uniquely identifies a particular instance of a `SpinLock` class. Furthermore, the waiting resource section (which uses `CheckWait` helper) is automatically also identified by the thread which executes it – this allows more than one thread to wait for the same spin lock.

<pre>mutex m; void thread0() { // Error: unique_lock lock(m); while (true) { do_work(); } } // unlock void thread1() { while (true) { unique_lock lock(m); do_other_work(); } // unlock }</pre>	<pre>mutex m; void thread0() { // Fixed: while (true) { unique_lock lock(m); do_work(); } // unlock } void thread1() { while (true) { unique_lock lock(m); do_other_work(); } // unlock }</pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

(a) A program with a nonterminating critical section (in `thread0`) and a deadlock (if `thread0` enters its critical section, `thread1` will wait infinitely). In C++ it is possible to use scope-based locks: the critical section belonging to mutex `m` is entered when `unique_lock lock(m)` is executed and left at the end of the scope in which the `lock` variable was defined (at the matching curly brace; also marked with comment `// unlock`).

(b) A fixed version of the program from Figure 6.4a (the start of the critical section was moved from the position `// Error` in the left code to `// Fixed` and therefore the critical section can end now). Intuitively, each critical section in this program terminates. However, as we can see in Figure 6.5, it is possible to find an infinite path in the state space of this program that infinitely waits for one of the critical sections. To make matters worse, this path can respect weak fairness.

Figure 6.4: Example programs with nonterminating resource section (a) and terminating resource sections (b).

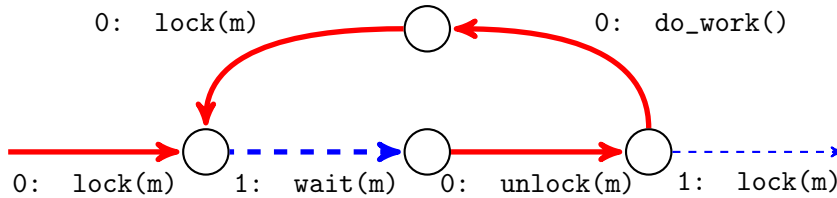


Figure 6.5: A fragment of state space of program in Figure 6.4b with starving lasso marked with bold edges. Each edge is marked by the thread it belongs to and the action of this thread. Furthermore, to ease the orientation, actions belonging to `thread0` are marked with continuous red edges while actions belonging to `thread1` are marked with dashed blue edges. We can see that both threads participate in the repeated part of the counterexample and `thread0` is denied the possibility (starves) to execute after `0: unlock(m)` (the thin blue dashed edge).

code might intuitively seem to terminate. First, `thread0` executes its `lock` action, then `thread1` starts waiting. If `thread0` always executes `unlock` and `lock` before `thread1` is allowed to run, `thread1` will never be able to finish waiting. The counterexample is illustrated in Figure 6.5 and is valid also under weak fairness assumptions.

In general, if a thread waits for some condition which is both infinitely often true and infinitely often false, there can be a run in which the waiting thread is only allowed to run at those moments when the condition is false. This type of runs is present in any program that uses busy waiting, which is very common in practice. For this practical reason, we cannot rely on the definition of nontermination as expressed with the LTL formula above, and we need a different way to describe nonterminating sections.

Definition 6.1: Nonterminating Resource Section Instance

A resource section instance is nonterminating if and only if it can reach a point from which it is not possible to reach its end.

For a particular resource section (e.g., waiting for `m` in `thread1`), checking for the absence of nonterminating resource section instances can be expressed using a CTL* property

$$\mathbf{AG} (wait-m-t1-start \implies \mathbf{A}[(\mathbf{EF} wait-m-t1-end) \mathbf{W} wait-m-t1-end])$$

(where \mathbf{W} is the weak until operator).

In general, the CTL* approach cannot be used, as it requires the set of resource sections to be known before the analysis starts, so that the formula can be created as a conjunction of formulas for each resource section. This is hard to do if resource sections can be created at runtime, which is often the case when dealing with programs in languages such as C and C++ – the number of objects such as threads, mutexes, or function invocations which are used to identify resource sections might be hard to determine without exploration of all the runs of the program.

6.4 Detection of Nontermination

The detection of nonterminating resource section instances in the context of explicit-state model checking proceeds as follows. The basic idea behind the detection of nonterminating resource section instances is that the model checker focuses on them one at a time. Every time a resource section instance is about to be entered during the state space exploration, the algorithm introduces a nondeterministic branching to the state space graph. In one branch, the resource section instance remains inactive, in which case the state space exploration proceeds as usual to discover other resource sections. However, in the other branch, the instance becomes active. Under this branch, the resource section instance is checked for being nonterminating – it becomes *active resource section instance* (ARSI). Note that the nondeterministic branching happens only outside of active resource sections, which means the ARSIs cannot be nested. Once the state space graph in the active branch reaches a state that is out of the scope of an ARSI, the state space exploration within this branch is stopped (a state with no successors is generated

outside the ARSI). Active resource section instances cannot be nested, but for any instance of a resource section nested in an active section instance, there is also an instance which is nested in an inactive section instance, and therefore can become active elsewhere in the state space. As a result of this construction, for every nonterminating resource section in the original program, there is a corresponding ARSI in the augmented state space graph. To let the exploration algorithm know that it is exploring a part of the state space that is within an ARSI, we mark all edges within ARSIs as accepting.³

³ The accepting label of an edge is a notation borrowed from (transition-based) Büchi automata. In DIVINE this information can be used by the exploration algorithm to behave differently for accepting edges.

An illustration of a state space graph augmented with nondeterministic choices and accepting edges is given in Figure 6.6. This augmentation of the state space can be performed by program instrumentation. Now to discover ARSIs which are nonterminating according to Definition 6.1, it is enough to detect terminal strongly connected components made of accepting edges only.

6.4.1 Detection Algorithm

Henceforward, we assume the state space graph is finite, and if a run of the program to be verified terminates then this fact is reflected by a state with no successors in the underlying state space graph. Note that the program may terminate even within a resource section instance. An ARSI terminates either by reaching the end of the section instance or by the termination of the whole underlying program. In both cases, this means a state with no successors is generated and reachable from the ARSI entrance point. Finally, we assume that any waiting is implemented in a nonblocking way; in particular, we require that waiting operations give rise to cycles in the state space of the waiting thread.⁴ As a result, the detection of nonterminating ARSIs can be performed as a search for an accepting terminal strongly connected component in the state space graph.

⁴ This is not a problem in practice as any blocking synchronisation (such as waiting for a mutex) can be simulated by a busy waiting loop.

Definition 6.2: Terminal Strongly Connected Component

A strongly connected component S is *terminal*⁵ if for each state v in S all successors of v are in S (there are no edges out of S).

⁵ Also sometimes called bottom strongly connected components, or closed communicating classes, especially in the area of probabilistic system analysis [Nor97].

Definition 6.3: Fully Accepting Terminal SCC (FATSCC)

A terminal strongly connected component of the state space is *fully accepting* (fully accepting terminal SCC, or FATSCC) if and only if it is nontrivial and all its edges are accepting.

Theorem 6.1. *A program contains a nonterminating resource section instance if and only if its state space graph contains a fully accepting terminal strongly connected component.*

Proof. Assume the program contains a nonterminating ARSI \mathcal{A} . Then there must exist a set of states in \mathcal{A} from which neither program end nor the corresponding resource section end can be reached. Among these states, there must be a subset which can be repeated indefinitely and cannot be left – a nontrivial terminal SCC which is part of an ARSI and therefore it is fully accepting – a FATSCC in the state space.

For the other direction, let us assume that there is a FATSCC in the state space graph. Since any edge which enters or leaves an ARSI is

```

mutex m1, m2;
{
  unique_lock l1(m1);
  do_work_1();
  {
    unique_lock l2(m2);
    do_work_2();
  } // unlock(m2)
} // unlock(m1)

```

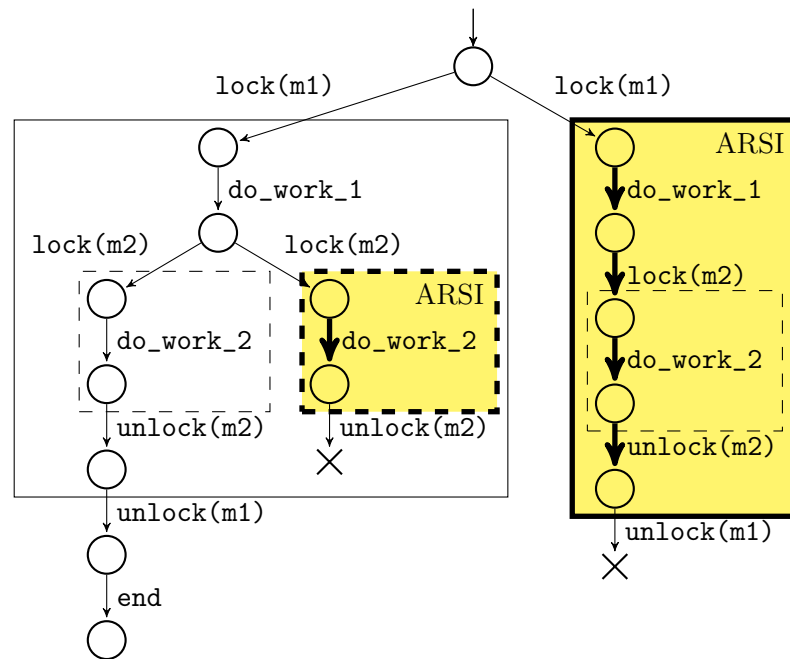


Figure 6.6: A small example of a program with two resource section instances (on the top) and its state space, which shows active resource section instances (ARSIs; on the bottom). In order to keep the state space simple, this example program is sequential and deterministic; the nondeterminism is caused only by the construction which gives rise to ARSIs. The resource section instances belonging to the critical section of mutex `m1` are wrapped in a solid rectangle in the image, while resource section instances belonging to `m2` are wrapped in a dashed rectangle. ARSIs are denoted by thick frame and yellow background and accepting edges in the state space are marked by thick arcs. Recall that active resource section instances cannot be nested. Crosses at the end of edges denote points where exploration of the state space was terminated due to reaching the end of an active resource section instance.

not accepting (which follows directly from the construction of the state space graph), all states that are part of the FATSCC must be states within a single ARSI. Since the component is terminal and nontrivial, it cannot be left. Furthermore, a program termination point cannot be part of the FATSCC as it has no successors and an ARSI end cannot be part the FATSCC as the edges going to it are not accepting. Therefore, it is impossible to reach either a program termination point or a state that would be outside of the resource section instance from the FATSCC, therefore, the FATSCC witnesses a resource section instance that does not terminate. \square

To detect the presence of a FATSCC in the state space graph, we employ the standard Tarjan's algorithm for finding strongly connected components. To decide if an SCC is terminal, it suffices to check that there are no edges going from it to any different SCC. Finally, to detect if a terminal component is nontrivial and fully accepting it is enough to check that the component contains at least one state with some successors (it is nontrivial) and that all states of the component have only accepting outgoing edges (it is fully accepting). These are minor modifications of the algorithm. Furthermore, it is possible to extend the algorithm to also perform safety checking while checking for nontermination – when a new edge with an error label is traversed, the exploration can be terminated immediately with a safety counterexample. This way, any need for separate safety checking is eliminated.

Note that it is also possible to define *global nontermination* using Definition 6.1. In this case, we only need to treat the whole program as a single active resource section instance.

6.4.2 Scheduling and Fairness

To provide further context, we also want to discuss the relation of our nontermination property to LTL model checking with fairness. Fairness constraints [BK08, Chapter 3.5] are needed in the analysis of temporal properties of parallel systems to avoid reporting of unrealistic counterexamples, such as those in which an enabled thread never gets the chance to make an action. However, even if we use LTL formula to describe nontermination and allow for LTL model checking under weak fairness, we still may obtain counterexamples that are totally unrealistic. This is because a weakly-fair scheduler⁶ admits runs in which the context switches that happen among participating threads are very regular, hence unrealistic.

The nontermination as defined in Definition 6.1 can be seen as a manifestation of an additional assumption about the thread scheduler. It claims that the scheduler is in essence somehow irregular, i.e., it will not allow for a context switch always after a fixed number of instructions or at a specific location in the code. Another way of looking at this is to assume that the scheduler is probabilistic and assigns some non-zero probability to interruption between any two instructions. With a probabilistic scheduler, we can equivalently define nonterminating resource section instance as a section instance which can get to the point when there is zero probability of reaching its end. Under the

[BK08] Baier et al., “Principles of Model Checking”.

⁶ For our purposes, a weakly-fair scheduler is a scheduler which ensures that on every accepting cycle in the state space all threads which existed (and were not blocked) during the execution of this cycle were also executed at least once on the cycle.

probabilistic view, we can also say that programs we denote as correct, i.e., without nonterminating sections, have zero probability of looping forever.

6.4.3 Implementation and Usage

We have implemented our nontermination detection approach in a branch of the DIVINE model checker. Resource sections can be specified by annotations in the source code of the program to be analysed by the user of the tool. Furthermore, DIVINE provides predefined resource sections for various POSIX thread (`pthread`) synchronisation primitives, namely for mutexes (including recursive and reader-writer mutexes), condition variables, barriers, and joining of threads. Since C++ threading support in DIVINE uses the `libc++` library which uses POSIX threads, these resource sections are also used for native C++ threading.

User-defined annotations can be given in one of the following categories: exclusive section, waiting for an event, and waiting for function end. For user-defined resource sections, DIVINE provides C and C++ interface which can be found on the web page accompanying this work.⁷ To make it possible to specify which resource section types should be considered for analysis, we use program instrumentation, which enables resource sections based on command line arguments (for more details see the accompanying web page). The instrumentation also ensures that edges which are part of an ARSI are accepting.

⁷<https://divine.fi.muni.cz/2019/Interm>

The detection of nonterminating resource sections in DIVINE uses Tarjan’s algorithm for finding strongly connected components. The algorithm runs on-the-fly, which means that it generates the state space graph as needed, and therefore, it can terminate before the entire state space graph is explored. The algorithm finishes if it finds a fully accepting terminal strongly connected component, if it discovers a safety error (to avoid the need for a separate safety verification), or once the entire state space is explored.

6.4.4 Interaction with Other Features of DIVINE

Since DIVINE is a research tool, not all the features implemented within the tool are expected to run together. In this case, there are some features of DIVINE which interfere with local nontermination detection in a not so obvious way.

Counterexamples When an error is found, DIVINE has support to show a counterexample and walk through it using an interactive simulator [Bar+17]. For safety properties, this counterexample is a sequence of states which ends with an error. For verification of properties described by LTL or Büchi automata (which are partially supported by DIVINE), the counterexample is a lasso-shaped trace. For nontermination, the part of the state space to be reported consists of a fully accepting terminal strongly connected component and a path that leads to it. However, it is not practical to output the information about the whole SCC, as it can be large. For this reason, DIVINE gives only a trace to

[Bar+17] Baranová et al., “Model Checking of C and C++ with DIVINE 4”.

the first state of the FATSCC (i.e., the first state from which end of the given resource section instance is not reachable).

Spurious Wakeups Condition variables are often used in parallel programs to block threads until some event occurs (e.g., a shared queue becomes non-empty). They provide a function which blocks the current thread (`wait`) and a function which signals the condition variable and causes waiting threads to proceed (`signal`). In most implementations, including C++ standard APIs and platform-specific APIs on Windows and Linux, `wait` is allowed to return before it is signalled: this behaviour is called *spurious wakeup* and programmers must take it into account when using condition variables.

To help with the discovery of bugs caused by spurious wakeup, DIVINE simulates spurious wakeup using nondeterministic choice. For nontermination detection, it is necessary to ensure that any spurious wakeup does not hide nontermination – we want to report resource section instances which can be only left by spurious wakeup as non-terminating. This can be done by careful implementation of the `wait` function in DIVINE – it first nondeterministically decides if a spurious wakeup will happen, and then, if it is not happening, it enters resource section which waits for `signal` and cannot be woken up spuriously. If the spurious wakeup is simulated, it behaves as if the thread was blocked and allows other threads to run. Once the waiting thread is used again for generation of successor states, it is unblocked and `wait` returns spuriously. The exhaustive enumeration of possible thread interleavings ensures that other threads can run arbitrarily long.

[LRB18] Lauko et al., “Symbolic Computation via Program Transformation”.

Data Nondeterminism and Symbolic Data To make it possible to verify programs that depend on input data, DIVINE has support for symbolic values [LRB18]. In an analysis of programs with symbolic values, the computation can be split when a branch depends on a symbolic value. This splitting can cause problems for nontermination detection if leaving some resource section instance requires a particular value of an input variable. Therefore, in the presence of symbolic data, nontermination checking might miss some instances of nontermination. We defer this problem to future work.

[ŠB18] Štill et al., “Model Checking of C++ Programs Under the x86-TSO Memory Model”.

Relaxed Memory Models DIVINE has support for analysis of parallel programs under the x86-TSO memory model of Intel and AMD CPUs ([ŠB18] and Chapter 5), which allows the program to exhibit behaviour not present under the interleaving semantics of threads. One of the main problems in interaction between nontermination and relaxed memory is that relaxed memory models over-approximate the possible behaviours of the system to cover all possibilities of contemporary and presumably also future processors of a given architecture. As nontermination is checking for *absence of termination*, it can spuriously hide nontermination if the state space of the program is over-approximated. Again, we defer this problem to future work.

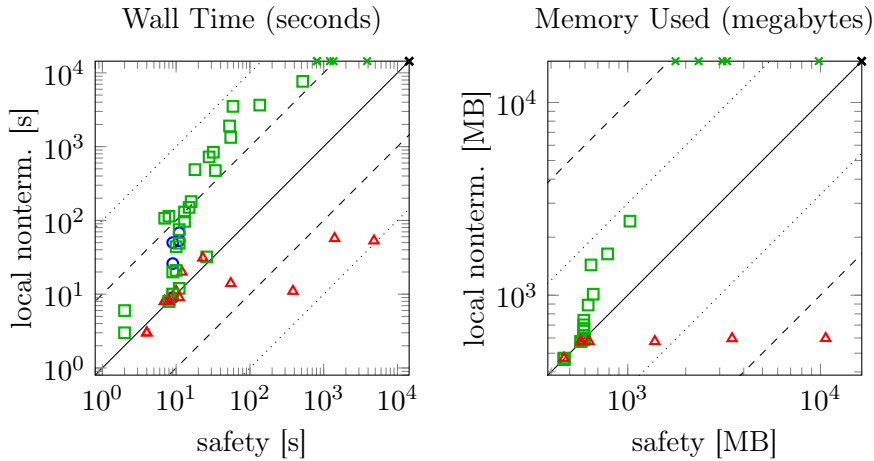


Figure 6.7: Scatter plots which compare local nontermination detection with safety checking as implemented in DIVINE. Both axes use a logarithmic scale. The dashed and dotted lines in wall time graphs signify $10\times$ and $100\times$ difference respectively. For graphs of memory usage, the dotted lines signify $3\times$ difference and the dashed $10\times$ difference. Green squares correspond to benchmarks which were error-less in both modes and blue circles correspond to benchmarks which contained errors in both cases. Red triangles correspond to benchmarks which contained a nonterminating section. The crosses on the outer edge of the plot correspond to timeouts and out-of-memory errors. All the failures for local/global nontermination were due to timeouts, benchmarks which failed with out-of-memory did so in all cases.

6.5 Evaluation

To our best knowledge, there is no suitable benchmark set that would cover termination in parallel programs. Therefore, we had to develop a suitable benchmark on our own. We naturally wanted to analyse the performance of our verification method on real-world data structures. Unfortunately, it is hard to reuse any existing real-world test cases of parallel data structures for verification, as these tests are usually developed as stress tests. Stress tests use large amounts of data and are supposed to be run for a long time in order to maximise a chance that a parallelism-related bug is found during the testing period. For the purpose of application of a formal verification tool such as DIVINE, the mentioned approach to testing of parallel programs is inappropriate. Since a model checker explores all interleavings of the program systematically within a single execution, further repeated executions, such as the ones within a stress test, are useless and only add to the complexity of the verification task. For these reasons, the tests we included in our benchmark are tests we created or adapted and modified specifically for the purpose of nontermination detection we wanted to evaluate.

To preserve some diversity at least, we opted for the following tests to be included in our benchmark. First, to cover some real-world scenarios, we created some tests for the Thread library from widely used C++ Boost⁸ (35 test cases). Second, we used some tests from

⁸ https://www.boost.org/doc/libs/1_69_0/doc/html/thread.html

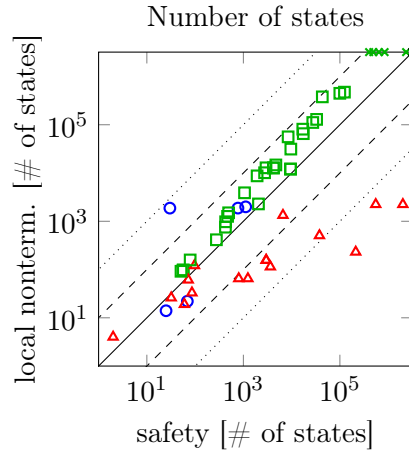


Figure 6.8: A comparison of state space sizes for local nontermination and safety. The dashed and dotted lines signify $10\times$ and $100\times$ difference respectively. The meaning of the marks in the graph is the same as in Figure 6.7.

DIVINE project itself (8 test cases), and finally, we developed a couple of specific tests for small programs demonstrating the behaviour of local nontermination with various synchronisation primitives (16 test cases). Overall, the benchmark covered usage of lock-free and mutex guarded parallel data structures (e.g. parallel queues), synchronised variables, less-used synchronisation primitives such as reader-writer locks, or a single-producer-single-consumer queue and the parallel hashset from [BRŠW15].

[BRŠW15] Barnat et al.,
“Fast, Dynamically-Sized
Concurrent Hash Table”.

To evaluate our verification approach, we let each test run with a 4 hours timeout and 16 GB memory limit. We measured runtime and memory requirements for the three following configurations of our tool:

safety A baseline configuration, in which the tool merely generates the state space of the program and checks for the standard safety issues, such as assertion violation, invalid memory access, etc. In this mode, no nontermination can be detected.

local nontermination The configuration in which the nonterminating resource section detection is used. Under this configuration, the state space of the original program is expanded with every entrance to the resource section as described in Section 6.3.

global nontermination The configuration that treats the whole program as a single resource section and detects if it terminates according to Definition 6.1. Since this configuration does not introduce additional nondeterminism, the state space of the program is roughly the same size as for *safety*.

The difference between local and global nontermination configurations is in the shape of the state space; both use the same algorithm (Tarjan’s algorithm for SCC decomposition). Thanks to this difference, local nontermination can be applied to programs which should not terminate, to check if each of its resource sections terminate.

Comparison of safety and local nontermination can be seen in Figure 6.7. We evaluate wall time and memory consumption – in practice, heavy-duty tools like DIVINE are likely to be used in long-running overnight tests (preferably only if anything relevant for the test changed since the last run), therefore longer runtimes might not be a big problem up to some point, but it is important to test that the verification tasks fit in some reasonable amount of memory. As we can see, the time overhead of local nontermination configuration is quite significant (up to $59\times$) especially for larger programs which are correct. As for memory consumption, we can see that total overhead is less than threefold, which is mostly due to the state space compression employed by DIVINE.

The wall time blow-up is due to extra nondeterminism introduced by active resource sections – the state space can grow by a factor that is related to the number of resource section instances encountered in the original state space. Note that many resource sections are likely to be very short. For programs that were invalid, i.e., contained some nonterminating resource sections, the verification usually exited faster under the local nontermination configuration than under the safety configuration, which means that once a nonterminating section is encountered, it is checked relatively quickly. Further insight into the comparison of safety and local nontermination can be seen in Figure 6.8, which compares sizes of state spaces for these two configurations. Here, we can see that the overhead in the size of the state space is lower than the time overhead (less than $10\times$). The extra time overhead is likely caused by inefficiencies in DIVINE. For example, when DIVINE nondeterministically chooses from N values, it will re-execute instructions between the last remembered state and the point of the nondeterministic choice N times.

Figure 6.9 shows a comparison of local nontermination with global nontermination and safety with global nontermination. Here, we can see that global nontermination behaves similarly to safety, with some time overhead caused by the somewhat more involved algorithm. This is well in line with our expectations, as global nontermination does not introduce any extra nondeterminism compared to safety and Tarjan’s algorithm runs in linear time with respect to the size of the state space, and so does reachability. This further highlights that the overwhelming part of the time overhead of local nontermination is in the increase of the state space size. It is important to note that local nontermination can be applied to programs which are intended to run infinitely (but have finite state space) – it can detect if there is a nonterminating resource section in such a program. As state space sizes and memory consumption are almost the same for safety and for global nontermination, we omit memory and state space size comparisons for the later two pairs of configurations.

Errors Found No errors were found in the C++ Boost tests. On the other hand, all the errors we artificially implanted in the test cases were found. As for the errors which were not deliberately introduced in the tests, we have found one error in a test of a lock-free queue from an older

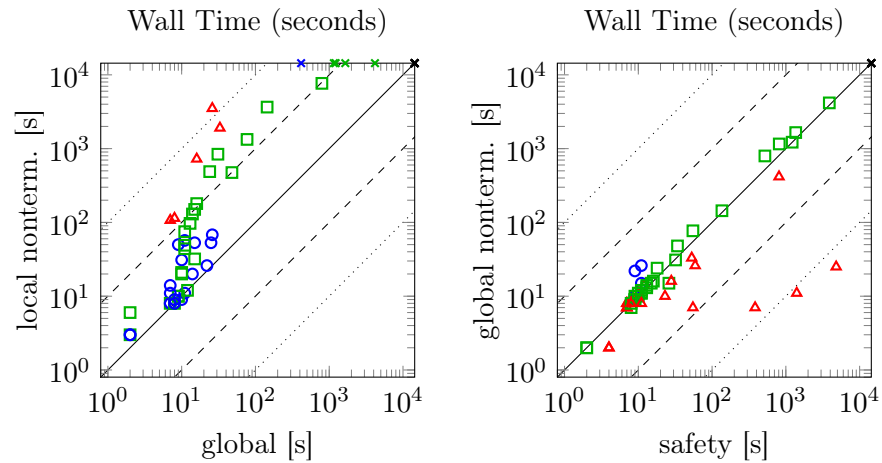


Figure 6.9: The first scatter plot compares local nontermination checking with checking if the whole program terminates (global nontermination). In this comparison, the red triangles correspond to benchmarks which did not end, but for which all resource sections terminated. Finally, in the second graph, we compare global nontermination checking with safety. Here, the red triangles correspond to benchmarks which did not terminate but were safe. See Figure 6.7 for the general description of the plot layouts.

version of DIVINE. The test was part of DIVINE’s test suite for a long time and was used to test that the queue works when it is continuously fed with elements while keeping its size bounded. This means that the test was deliberately nonterminating and the intention was that all the operations executed by main loops of the test’s two threads terminate, which was not the case – a variable which was supposed to keep track of the size of the queue was not maintained properly, and therefore it could have happened that the reader thread would wait indefinitely, attempting to read from an empty queue which would never fill up. So far, the test case was run under DIVINE with the safety algorithm only, therefore, the error did not manifest and remained undetected.

6.6 Related Work

For the related work, we consider only results which go beyond safety checking. There are many approaches to find problems such as assertion violations or memory safety violations, but they are often fundamentally limited to properties concerning finite runs of the program, and we are focusing here on an infinite behaviour, namely on the absence of termination. Similarly, we do not explore in depth techniques which specialise on checking sequential programs and have no support for parallelism, as well as techniques which are tailored to a specific modelling language and cannot be applied in general.

Several techniques for checking properties other than safety exist – indeed usage of various temporal logics, such as Linear Temporal Logic (LTL) [BK08, Chapter 5] and Computation Tree Logic (CTL) [BK08, Chapter 6] in the context of model checking dates way back

[BK08] Baier et al., “Principles of Model Checking”.

to the beginning of the research of formal methods. Unfortunately, these techniques are not often applied to programs written in real-world programming languages such as C and C++.

As for techniques which detect nontermination, both static and dynamic techniques exist for the detection of deadlocks caused by circular waiting for mutexes [CC14; Aga+10; BH05]. These techniques specialise on mutexes and do not allow general nontermination detection, and it is unlikely that they could be naturally extended to cover it. Some methods detect deadlocks of the whole program (i.e., a program state from which the program cannot move) [Cha+05; DIS99], but these techniques cannot find cases in which only some threads of the program are making progress, while other threads are blocked forever. Also, these global deadlock detection techniques are inadequate in the presence of synchronisation mechanisms which causes busy waiting instead of blocking (for example spin locks) or in the cases when normally blocking operations are implemented using busy waiting (which can be easier to handle for the verifier in some cases). A somewhat different approach based on communicating channels is proposed in [NY16], but this approach is aiming at the Go programming language which primarily uses shared channels for communication between threads. Overall, neither of these techniques is applicable in general for the detection of nontermination in programs which use a combination of synchronisation primitives in shared memory. There are also techniques for checking termination of sequential programs [CPR06; BCDO06; DHLP15; Bro+16; Hen+17; Gie+17] and techniques for termination analysis in parallel programs that target modelling languages [KF14; AFGM17].

A more realistically-targeted termination checker for parallel programs is presented in [CPR07]. Its primary goal is to show a given thread terminates. It abstracts other threads into a model of the environment which is incrementally refined and attempts to prove termination of the thread locally. It appears to support C programs, but it assumes the number of threads is fixed and it does not support synchronisation with mutexes and similar primitives. Another tool for termination-checking parallel C programs is presented in [PR12], it is similar to [CPR07] but more general. It uses transition invariants and well-foundedness to prove termination of whole programs or individual threads and uses predicate abstraction and refinement. The downside of this approach is that it can report unfair paths as termination counterexamples.

Probably the closes relative to our approach is shown in [ABEL12]. It presents a tool MUTANT that can detect fair nontermination. It aims to identify ultimately periodic executions (i.e., executions in which the same sequence of states is repeated, this sequence can be preceded by a finite stem) and therefore is complete only in programs with finite state space. Furthermore, the number of context switches in the stem and each of the periods is bounded. The tool uses sequentialisation – it reduces the problem of (context-bounded) termination checking of a parallel program to assertion checking in a sequential program. It does not target any realistic programming language; however, it targets the Boogie modelling language for which translators from C and other

[CC14] Cai et al., “Magi-clock: Scalable Detection of Potential Deadlocks in Large-Scale Multithreaded Programs”.

[Aga+10] Agarwal et al., “Detection of Deadlock Potentials in Multithreaded Programs”.

[BH05] Bensalem et al., “Scalable dynamic deadlock analysis of multi-threaded programs”.

[Cha+05] Chaki et al., “Concurrent software verification with states, events, and deadlocks”.

[DIS99] Demartini et al., “A deadlock detection tool for concurrent Java programs”.

[NY16] Ng et al., “Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis”.

[CPR06] Cook et al., “Termination Proofs for Systems Code”.

[BCDO06] Berdine et al., “Automatic Termination Proofs for Programs with Shape-Shifting Heaps”.

[DHLP15] Dietsch et al., “Fairness Modulo Theory: A New Approach to LTL Software Model Checking”.

[Bro+16] Brockschmidt et al., “T2: Temporal Property Verification”.

[Hen+17] Hensel et al., “AProVE: Proving and Disproving Termination of Memory-Manipulating C Programs”.

[Gie+17] Giesl et al., “Analyzing Program Termination and Complexity Automatically with AProVE”.

[KF14] Kupriyanov et al., “Causal Termination of Multi-threaded Programs”.

[AFGM17] Albert et al., “Rely-guarantee termination and cost analyses of loops with concurrent interleavings”.

[CPR07] Cook et al., “Proving Thread Termination”.

[PR12] Popeea et al., “Compositional Termination Proofs for Multithreaded Programs”.

[ABEL12] Atig et al., “Detecting Fair Non-termination in Multithreaded Programs”.

languages exist. Therefore, it might be possible to use this tool even for C and other programming languages. The main difference between our approach and the one in [ABEL12] is that our approach uses a different notion of fairness, targets primarily C and C++, does not impose any limits on the number of context switches, and can detect local nontermination.

6.7 Conclusion

We have presented a novel approach to detection of parts of real-world programs written in C and C++ which do not terminate. Our method allows for detection of partial deadlocks (and livelocks) caused by misuse of synchronisation, but it is not limited to any particular mode of parallel programming (such as lock-based synchronisation, or programs with communication channels) and indeed allows any combination of synchronisation allowed by C++ itself. To achieve this, it is necessary to provide simple annotations for parts of the code which are to be checked for termination. Our implementation in the DIVINE model checker ships with these annotations already prepared for verification of programs which use C++ blocking synchronisation primitives (mutexes, condition variables), or similar synchronisation primitives from the POSIX threads library (`pthread`s). Due to the universality of these synchronisation primitives, our annotations allow for checking of most programs which use blocking synchronisation out of the box. For lock-free programs, or custom synchronization primitives, users have to annotate functions or blocks of code which are required to be exited once they were entered.

We have implemented our technique in an open-source model checker DIVINE and evaluated it on a set of benchmarks including our tests of the Thread library from widely used C++ Boost. The evaluation shows that while the time overhead of local nontermination checking can be quite significant (up to $59\times$ compared to safety checking on our benchmarks), the memory overhead is quite modest (under $3\times$). During the evaluation, we have discovered a hidden bug that remained in the code for a couple of years, even though the code was subject to intensive safety checking.

Our technique enables checking nontermination in parallel programs, including detection of partial deadlocks and livelocks. It also supports detection of cases when infinitely-running programs contain sections which are supposed to terminate but do not terminate. We believe that even the overhead shown in our evaluation is worth paying for the additional guarantees over safety checking. While related to verification of properties written in temporal logics such as CTL*, our technique cannot be subsumed into CTL* verification, as CTL* cannot quantify over objects which can be created while the program runs.

For future work, it is crucial to further investigate interactions between nontermination checking and relaxed memory, and nontermination and symbolic data representation, as the presence of either of these features can lead to programs being reported as terminating even if they are not in the current situation. Nevertheless, even in

the presence of relaxed memory or symbolic data, any reported non-terminating section of the program is indeed a case when the program cannot proceed past the given point. We would also like to investigate better algorithms for detection of local nontermination that might avoid adding nondeterminism to the program under analysis.

Chapter 7

Conclusion

In this work, we have introduced the problem of analysis of parallel C++ programs and more generally of analysis of parallel programs written in real-world programming languages. We have focused on the analysis techniques that can help programmers to discover hard-to-find bugs (caused by thread interaction or other problems such as mishandling of memory). Furthermore, we have aimed at techniques that can be realistically used by programmers (i.e., do not require a separate modelling effort before the analysis). We have performed a review of the state of the art in the area of analysis of parallel programs written in realistic programming languages and contributed to it in three main areas. First, in Chapter 4, we took a look at the problem of analysis of programs in a high-level programming language (C++ in our case), in particular, we focused on the addition of support for C++ exceptions to the DIVINE model checker. Second, in Chapter 5, we proposed a novel technique for analysis of programs under the relaxed memory model of x86-64 processors. Finally, in Chapter 6, we proposed an automatic technique for finding parts of a finite-state parallel program which should terminate but do not terminate. Overall, we believe our contributions improve on the state of the art in the analysis of parallel programs and offer utility to both researchers as well as developers.

7.1 Contributions

Language Support In the area of language support, we have shown that reuse of existing libraries (which are not designed with program analysis in mind) is a valid option that can both save effort and improve analysis fidelity. In particular, we have focused on the case of exception support for C++, and we have devised a solution which allows us to reuse existing exception-handling code from the C++ standard library implementation. This approach required only minimal modifications of the DIVINE verifier and a small new library which implements platform-specific stack unwinding routines for DIVINE and is linked to the analysed program. These results contrast with a previous attempt to support exceptions in an older version of DIVINE that required more complex changes to the core of the verifier and had to re-implement exception matching. We have also shown that our solution can handle cases which were supported correctly in neither the older DIVINE nor

ESBMC, which is one of the few other tools with reasonable support for C++ exceptions. While the core of the work presented in this chapter is about C++ exceptions, we believe its message is more general, and reuse of existing libraries is a valid approach for extension of language support in program analysis tools. Indeed, DIVINE reuses the entire C++ standard library with only minor modifications, and its C standard library is also in large part reused.

Relaxed Memory In the area of analysis of programs running under relaxed memory models, we have proposed a novel operational semantics for the x86-TSO memory model of common Intel and AMD x86-64 processors. This operational semantics allows us to lower the amount of nondeterminism introduced by the simulation of relaxed behaviour, and therefore, it reduces the overall complexity of program analysis. The crucial idea behind this method is that it moves the nondeterminism to the load (and fence) operations. This delayed nondeterminism allows us to reduce the number of redundant runs that are simulated. Our technique can impose bounds on the number of operations that can be delayed by each thread, in which case it can be applied to any program which has finite state space under sequential consistency. It can also forego the buffer bound, in which case it might not terminate for programs which do not terminate even if they have finite state space under sequential consistency.

Local Nontermination The final area of our work was checking non-termination in parallel programs. We have devised a method based on state-space exploration that can prove termination or nontermination of programs with finite state space. This approach is significantly different from most existing methods for proving termination or nontermination. These methods usually focus on sequential programs or, in the rare case they support parallel programs, they typically focus on thread-modular proofs, i.e., proofs that perform reasoning on each thread separately while abstracting the other threads. While this allows these competing techniques to support programs with possibly infinite state spaces, it also makes them more complex, which is witnessed by the fact that tools which implement these techniques for programming languages and support parallelism are rare. In our work, we have therefore limited ourselves to programs with finite state space – our analysis procedure will not stop if the state space is infinite unless it happens to find an error with a finite witness. However, our technique is reasonably simple to implement and can be readily applied to C and C++ programs. On top of that, our technique can detect *local nontermination*; i.e., it can detect that a part of the program which is supposed to terminate does not. Such parts can include a critical section or a function of a thread-safe data structure, for example, a `pop` operation of a queue. This ability makes our technique useful for analysis of event loops and similar constructs that can run without termination but dispatch procedures that must terminate.

Implementation and Evaluation The three main contributions presented in this thesis are accompanied by an open-source implementation in the DIVINE model checker and are evaluated.

7.2 Future Work

Both the program analysis methods and the programming languages they are targeting are under steady development. We will now shortly focus on some of the directions which could improve on the results presented in this thesis.

Language Support The C++ language support in DIVINE is mostly complete for the C++17 standard. With the soon-to-be-released C++20 standard, we expect that most of the support will be covered by updates of clang compiler and the C++ standard library we reuse, and therefore will require little effort from the DIVINE developers. A more ambitious future work would be the addition of support of other programming languages to DIVINE. Currently, DIVINE supports C and C++, but there are many other languages which can be compiled to LLVM, and therefore it should be possible to integrate them with DIVINE. Such an addition would have two important consequences. First, it would either provide further validation to the general approach of component reuse in program analysis or discover some of its possible limitations. Second, it would extend the number of programs in which DIVINE can discover problems.

Finally, one of the missing features of DIVINE concerning C++ is support for the C++ relaxed memory model.

Relaxed Memory There are three main directions in which future work on relaxed memory in DIVINE could continue. First, it would be useful to perform a new comprehensive evaluation which would compare our method with recent methods build on stateless model checking that appeared around the time of publication of our approach or after it. Such an evaluation would shed further light on the promises of stateless and explicit-state model checking and could inform the further general approach to the analysis of relaxed memory models in our tool.

Second, if our approach continues to hold promises compared to the new developments by other researches, or if it can be extended to at least match them, it remains to be seen how it can be extended to more relaxed memory models, for example, the ARM memory model or the memory model of C++. These memory models are more complex both in the sense that the number of possible executions under them is higher, which worsens the state space explosion problem and in the sense that the behaviour itself is more complex which makes it harder to simulate them.

Finally, it remains to be seen if the program transformation used in our x86-TSO support can be efficiently combined with the program transformation that introduces symbolic and abstract data as presented in [LRB18], without the need to integrate the two transformations tightly.

[LRB18] Lauko et al., “Symbolic Computation via Program Transformation”.

Termination Checking Termination checking of parallel programs is a topic which is not widely explored thus far, at least compared to the area of relaxed memory models or termination checking of sequential programs. Some of the areas in which our method could be extended are apparent from its current limitations – the future work should include integration with analysis under the x86-TSO memory model and analysis of programs with symbolic (or abstract) data. It is also likely that a more efficient algorithm for detection of local nontermination can be devised.

Further research area would be to investigate the practical performance difference between our method and the existing methods based mostly on thread-modular proofs and well-founded relations. Currently, we have a hypothesis that our method would offer advantages in cases where the complexity of the program arises mainly from thread interactions, while the thread-modular approaches would handle programs with complex data manipulations better. After that, it might be interesting to investigate possibilities to combine our method with existing research in this area. For example, it might be possible to use our method to discover potential nontermination in abstracted state space and then use arguments based on well-foundedness to attempt to validate the counterexample and refine the abstraction.

Appendix A

Published Papers

In this appendix, I summarize my publication results in the field of program analysis. For my most important papers, I also include a short description of the paper and a description of my contribution to the paper. The remainder of papers are mostly either older papers to which I have contributed less or short report papers for the Software Verification Competition (SV-COMP [Bey20]).

[Bey20] Beyer, “Advances in Automatic Software Verification: SV-COMP 2020”.

A.1 Most Significant Papers

Local Nontermination Detection for Parallel C++ Programs

In this paper, we present our approach to ensuring that parallel programs do not hang or wait indefinitely – i.e., there are no deadlocks, livelocks, and the program proceeds towards its goals. The paper contains the theoretical description of our approach and evaluation of our publicly available implementation.

My Contribution: The algorithm design, implementation and writing of the paper was done by me, my advisor Jiří Barnat helped by consulting the theory with me and proofreading the paper. I have presented this paper on the SEFM 2019 conference. 90 %

Vladimír Štill and Jiří Barnat. “Local Nontermination Detection for Parallel C++ Programs”. In: *Software Engineering and Formal Methods*. Cham: Springer International Publishing, 2019, pp. 373–390. ISBN: 978-3-030-30446-1. DOI: 10.1007/978-3-030-30446-1_20. URL: <https://divine.fi.muni.cz/2019/lnterm> [ŠB19]

Model Checking of C++ Programs Under the x86-TSO Memory Model

Here we present a novel approach to verification of parallel programs with respect to the memory model of Intel processors. The approach improves the efficiency of explicit-state model checking by decreasing amount of nondeterminism in the program. The paper contains evaluation and is accompanied by a publicly available implementation.

My Contribution: The algorithm design, implementation and writing of the paper was done by me, Jiří Barnat helped by consulting the

theory with me and proofreading the paper. I have presented this paper on the ICFEM 2018 conference. 90 %

Vladimír Štill and Jiří Barnat. “Model Checking of C++ Programs Under the x86-TSO Memory Model”. In: *Formal Methods and Software Engineering*. Cham: Springer International Publishing, 2018, pp. 124–140. ISBN: 978-3-030-02450-5. DOI: 10.1007/978-3-030-02450-5_8. URL: <https://divine.fi.muni.cz/2018/x86tso> [ŠB18]

Using Off-the-Shelf Exception Support Components in C++ Verification

In this paper, we present an extension of DIVINE that allows it to verify programs that contain C++ exceptions and C programs with a non-local transfer of control flow (`setjmp/longjmp`). We show that with careful design, we can successfully reuse exception handling code from the standard C++ library. The result is that virtually any exception handling constructs working in the standard C++ are now available in DIVINE.

My Contribution: I have designed the exception support for DIVINE 4, implemented it and performed the evaluation for this paper. I have also written most of the text for the paper. Petr Ročkai and Jiří Barnat helped by consulting the design and implementation and also helped with the paper text. I have presented this paper on the QRS 2017 conference. The paper and its presentation were awarded the best paper award. 75 %

Vladimír Štill, Petr Ročkai, and Jiří Barnat. “Using Off-the-Shelf Exception Support Components in C++ Verification”. In: *IEEE International Conference on Software Quality, Reliability and Security (QRS)*. July 2017, pp. 54–64. DOI: 10.1109/QRS.2017.15. URL: <https://divine.fi.muni.cz/2017/exceptions/> [ŠRB17]

Model Checking of C and C++ with DIVINE 4

In this tool paper, we describe the overall architecture of DIVINE 4 and changes in the tool compared to DIVINE 3. Most significantly, this paper describes the modular nature of DIVINE: DIVINE 4 is built around an efficient interpreter which, together with a small, verification-oriented operating system and a set of runtime libraries, enables verification of real-world code written in C and C++.

My Contribution: The text of the paper is written mostly by me, with additions and proofreading by Petr Ročkai and Jiří Barnat. The architecture design was mostly due to Petr Ročkai, with additions by me and Jan Mrázek. The implementation includes code by all the co-authors with most significant contributions by (in the order of significance) Petr Ročkai, me, and Jan Mrázek. 30 %

Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Heinrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. “Model Checking of C and C++ with DIVINE 4”. In: *International Symposium on*

Automated Technology for Verification and Analysis (ATVA). vol. 10482. Lecture Notes in Computer Science. 2017. DOI: 10.1007/978-3-319-68167-2_14. URL: <https://divine.fi.muni.cz/2017/divine4/> [Bar+17]

DiVM: Model Checking with LLVM and Graph Memory

This paper introduces the concept of a virtual machine with graph memory as a core component for explicit-state and abstraction-based verification of software. The paper is accompanied by an implementation of the virtual machine which runs LLVM IR (which can be obtained from C or C++ using the clang compiler) and an evaluation which compares the new approach to a more traditional design of an LLVM-based model checker as well as a symbolic model checker.

My Contribution: The primary author of this paper is Petr Ročkai. My contribution concerned mostly the C++ support (including program compilation and libraries) and the evaluation and comparison of the new approach with DIVINE 3 and ESBMC. 20%

Petr Ročkai, Vladimír Štill, Ivana Černá, and Jiří Barnat. “DiVM: Model checking with LLVM and graph memory”. In: *Journal of Systems and Software* 143 (2018), pp. 1–13. DOI: 10.1016/j.jss.2018.04.026. URL: <https://divine.fi.muni.cz/2017/divm/> [RŠČB18]

A.2 Other Papers

Henrich Lauko, Vladimír Štill, Petr Ročkai, and Jiří Barnat. “Extending DIVINE with Symbolic Verification Using SMT”. in: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, pp. 204–208. ISBN: 978-3-030-17502-3. DOI: 10.1007/978-3-030-17502-3_14 [LŠRB19]

Jan Mrázek, Martin Jonáš, Vladimír Štill, Henrich Lauko, and Jiří Barnat. “Optimizing and Caching SMT Queries in SymDIVINE”. in: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2017, pp. 390–393. ISBN: 978-3-662-54580-5. DOI: 10.1007/978-3-662-54580-5_29 [Mrá+17]

Vladimír Štill, Petr Ročkai, and Jiří Barnat. “DIVINE: Explicit-State LTL Model Checker”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2016, pp. 920–922. ISBN: 978-3-662-49674-9. DOI: 10.1007/978-3-662-49674-9_60 [ŠRB16a]

Jiří Barnat, Ivana Černá, Petr Ročkai, Vladimír Štill, and Kristína Zákopčanová. “On Verifying C++ Programs with Probabilities”. In: *ACM Symposium on Applied Computing*. 2016, pp. 1238–1243. ISBN: 978-1-4503-3739-7. DOI: 10.1145/2851613.2851721 [Bar+16]

Vladimír Štill, Petr Ročkai, and Jiří Barnat. “Weak Memory Models as LLVM-to-LLVM Transformations”. In: *Mathematical and Engineering Methods in Computer Science, Revised Selected Papers*. Vol. 9548. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 144–155. ISBN: 978-3-319-29817-7. DOI: 10.1007/978-3-319-29817-7_13 [ŠRB16]

Jiří Barnat, Petr Ročkai, Vladimír Štill, and Jiří Weiser. “Fast, Dynamically-Sized Concurrent Hash Table”. In: *Model Checking Software (SPIN 2015)*. Vol. 9232. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 49–65. ISBN: 978-3-319-23403-8. DOI: 10.1007/978-3-319-23404-5_5 [BRŠW15]

Petr Ročkai, Vladimír Štill, and Jiří Barnat. “Techniques for Memory-Efficient Model Checking of C and C++ Code”. In: *Software Engineering and Formal Methods*. Vol. 9276. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 268–282. ISBN: 978-3-319-22968-3. DOI: 10.1007/978-3-319-22969-0_19 [RŠB15]

Vladimír Štill, Petr Ročkai, and Jiří Barnat. “Context-Switch-Directed Verification in DIVINE”. in: *Mathematical and Engineering Methods in Computer Science*. Vol. 8934. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 135–146. ISBN: 978-3-319-14895-3. DOI: 10.1007/978-3-319-14896-0_12 [ŠRB14]

Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. “DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs”. In: *Computer Aided Verification*. Vol. 8044. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 863–868. ISBN: 978-3-642-39798-1. DOI: 10.1007/978-3-642-39799-8_60 [Bar+13]

Appendix B

Bibliography

- [AABN17] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. “Context-Bounded Analysis for POWER”. In: *TACAS*. Berlin, Heidelberg: Springer, 2017, pp. 56–74. DOI: 10.1007/978-3-662-54580-5_4.
- [AAJL16] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. “Stateless Model Checking for POWER”. In: *Computer Aided Verification*. Cham: Springer International Publishing, 2016, pp. 134–156. ISBN: 978-3-319-41540-6. DOI: 10.1007/978-3-319-41540-6_8.
- [AAJN18] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. “Optimal Stateless Model Checking under the Release-Acquire Semantics”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276505. URL: <https://doi.org/10.1145/3276505>.
- [AAJS14] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. “Optimal Dynamic Partial Order Reduction”. In: *POPL*. San Diego, California, USA: ACM, 2014, pp. 373–384. DOI: 10.1145/2535838.2535845.
- [ABBM10] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. “On the Verification Problem for Weak Memory Models”. In: *POPL*. Madrid, Spain: ACM, 2010, pp. 7–18. DOI: 10.1145/1706299.1706303.
- [Abd+15] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. “Stateless Model Checking for TSO and PSO”. In: *TACAS*. Berlin, Heidelberg: Springer, 2015, pp. 353–367. DOI: 10.1007/978-3-662-46681-0_28.
- [Abd+17] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. “Stateless model checking for TSO and PSO”. In: *Acta Informatica* 54.8 (2017), pp. 789–818. DOI: 10.1007/s00236-016-0275-0.
- [Abd+19] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. “Optimal Stateless Model Checking for Reads-from Equivalence under Sequential Consistency”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360576. URL: <https://doi.org/10.1145/3360576>.
- [ABEL12] Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal. “Detecting Fair Non-termination in Multithreaded Programs”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 210–226. ISBN: 978-3-642-31424-7. DOI: 10.1007/978-3-642-31424-7_19.
- [ABP11] Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. “Getting Rid of Store-Buffers in TSO Analysis”. In: *CAV*. Berlin, Heidelberg: Springer, 2011, pp. 99–115. DOI: 10.1007/978-3-642-22110-1_9.

- [AFGM17] Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. “Rely-guarantee termination and cost analyses of loops with concurrent interleavings”. In: *Journal of Automated Reasoning* 59.1 (2017), pp. 47–85. DOI: 10.1007/s10817-016-9400-6.
- [Aga+10] Rahul Agarwal, Saddek Bensalem, Eitan Farchi, Klaus Havelund, Yarden Nir-Buchbinder, Scott D Stoller, Shmuel Ur, and Liqiang Wang. “Detection of Deadlock Potentials in Multithreaded Programs”. In: *IBM Journal of Research and Development* 54.5 (2010), pp. 3–1. DOI: 10.1147/JRD.2010.2060276.
- [AJLS18] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. “Optimal Dynamic Partial Order Reduction with Observers”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2018, pp. 229–248. ISBN: 978-3-319-89963-3. DOI: 10.1007/978-3-319-89963-3_14.
- [AKNT13] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. “Software Verification for Weak Memory via Program Transformation”. In: *ESOP*. Berlin, Heidelberg: Springer, 2013, pp. 512–532. DOI: 10.1007/978-3-642-37036-6_28.
- [AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. “Partial Orders for Efficient Bounded Model Checking of Concurrent Software”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 141–157. ISBN: 978-3-642-39799-8. DOI: 10.1007/978-3-642-39799-8_9.
- [Alb+17] Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. “Context-Sensitive Dynamic Partial Order Reduction”. In: *Computer Aided Verification*. Cham: Springer International Publishing, 2017, pp. 526–543. ISBN: 978-3-319-63387-9. DOI: 10.1007/978-3-319-63387-9_26.
- [Alb+19] Elvira Albert, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, Miguel Isabel, and Peter J. Stuckey. “Optimal Context-Sensitive Dynamic Partial Order Reduction with Observers”. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSA 2019. Beijing, China: Association for Computing Machinery, 2019, pp. 352–362. ISBN: 9781450362245. DOI: 10.1145/3293882.3330565. URL: <https://doi.org/10.1145/3293882.3330565>.
- [AM11] Jade Alglave and Luc Maranget. “Stability in Weak Memory Models”. In: *CAV*. Berlin, Heidelberg: Springer, 2011, pp. 50–66. DOI: 10.1007/978-3-642-22110-1_6.
- [AMK18] Pavel Andrianov, Vadim Mutilin, and Alexey Khoroshilov. “Predicate Abstraction Based Configurable Method for Data Race Detection in Linux Kernel”. In: *Tools and Methods of Program Analysis*. Cham: Springer International Publishing, 2018, pp. 11–23. ISBN: 978-3-319-71734-0. DOI: 10.1007/978-3-319-71734-0_2.
- [AMSS10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. “Fences in Weak Memory Models”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 258–272. DOI: 10.1007/978-3-642-14295-6_25.
- [AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. “Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory”. In: *ACM Transactions on Programming Languages and Systems* 36.2 (July 2014), 7:1–7:74. ISSN: 0164-0925. DOI: 10.1145/2627752.
- [And+17] Pavel Andrianov, Karlheinz Friedberger, Mikhail Mandrykin, Vadim Mutilin, and Anton Volkov. “CPA-BAM-BnB: Block-Abstraction Memoization and Region-Based Memory Models for Predicate Abstractions”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 355–359. ISBN: 978-3-662-54580-5. DOI: 10.1007/978-3-662-54580-5_22.

- [AV19] Cyrille Artho and Willem Visser. “Java Pathfinder at SV-COMP 2019 (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, pp. 224–228. ISBN: 978-3-030-17502-3. DOI: 10.1007/978-3-030-17502-3_18.
- [BAM07] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. “CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 12–21. ISBN: 9781595936332. DOI: 10.1145/1250734.1250737.
- [Ban+16] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Păsăreanu, and Tevfik Bultan. “String Analysis for Side Channels with Segmented Oracles”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 193–204. ISBN: 9781450342186. DOI: 10.1145/2950290.2950362.
- [Bar+13] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. “DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs”. In: *Computer Aided Verification*. Vol. 8044. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 863–868. ISBN: 978-3-642-39798-1. DOI: 10.1007/978-3-642-39799-8_60.
- [Bar+16] Jiří Barnat, Ivana Černá, Petr Ročkai, Vladimír Štill, and Kristína Zákopčanová. “On Verifying C++ Programs with Probabilities”. In: *ACM Symposium on Applied Computing*. 2016, pp. 1238–1243. ISBN: 978-1-4503-3739-7. DOI: 10.1145/2851613.2851721.
- [Bar+17] 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: *International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Vol. 10482. Lecture Notes in Computer Science. 2017. DOI: 10.1007/978-3-319-68167-2_14. URL: <https://divine.fi.muni.cz/2017/divine4/>.
- [Bat+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. “Mathematizing C++ Concurrency”. In: *Principles of Programming Languages*. POPL ’11. Austin, Texas, USA: ACM, 2011, pp. 55–66. DOI: 10.1145/1926385.1926394.
- [BBČŠ05] Jiří Barnat, Luboš Brim, Ivana Černá, and Pavel Šimeček. “DiVinE – The Distributed Verification Environment”. In: *Proceedings of 4th International Workshop on Parallel and Distributed Methods in verification*. July 2005, pp. 89–94.
- [BBR12] Jiří Barnat, Luboš Brim, and Petr Ročkai. “Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs”. In: *NASA Formal Methods: 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 252–266. ISBN: 978-3-642-28891-3. DOI: 10.1007/978-3-642-28891-3_25.
- [BBW14] Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. “Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR)”. In: *Computer Aided Verification*. Cham: Springer International Publishing, 2014, pp. 831–848. ISBN: 978-3-319-08867-9. DOI: 10.1007/978-3-319-08867-9_55.
- [BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. “Symbolic Model Checking without BDDs”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207. ISBN: 978-3-540-49059-3. DOI: 10.1007/3-540-49059-0_14.

- [BCDM15] Ahmed Bouajjani, Georgel Calin, Egor Derevenetc, and Roland Meyer. “Lazy TSO Reachability”. In: *FASE*. Berlin, Heidelberg: Springer, 2015, pp. 267–282. DOI: 10.1007/978-3-662-46675-9_18.
- [BCDO06] Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O’Hearn. “Automatic Termination Proofs for Programs with Shape-Shifting Heaps”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 386–400. ISBN: 978-3-540-37411-4. DOI: 10.1007/11817963_35.
- [BD20] Dirk Beyer and Matthias Dangl. “Software Verification with PDR: An Implementation of the State of the Art”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2020, pp. 3–21. ISBN: 978-3-030-45190-5. DOI: 10.1007/978-3-030-45190-5_1.
- [BDM13] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. “Checking and Enforcing Robustness against TSO”. In: *ESOP*. Berlin, Heidelberg: Springer, 2013, pp. 533–553. DOI: 10.1007/978-3-642-37036-6_29.
- [Bey17] Dirk Beyer. “Software Verification with Validation of Results”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 331–349. DOI: 10.1007/978-3-662-54580-5_20.
- [Bey20] Dirk Beyer. “Advances in Automatic Software Verification: SV-COMP 2020”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2020, pp. 347–367. ISBN: 978-3-030-45237-7. DOI: 10.1007/978-3-030-45237-7_21.
- [BH05] Saddek Bensalem and Klaus Havelund. “Scalable dynamic deadlock analysis of multi-threaded programs”. In: *Parallel and Distributed Systems: Testing and Debugging 2005* (2005). URL: https://www.researchgate.net/profile/Klaus_Havelund/publication/221471856_Dynamic_Deadlock_Analysis_of_Multi-threaded_Programs/links/02bfe51368e93ccaa4000000.pdf.
- [BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN: 978-0-262-02649-9.
- [BK11] Dirk Beyer and M. Erkan Keremoglu. “CPAchecker: A Tool for Configurable Software Verification”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 184–190. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_16.
- [BM08] Sebastian Burckhardt and Madanlal Musuvathi. “Effective Program Verification for Relaxed Memory Models”. In: *CAV*. Berlin, Heidelberg: Springer, 2008, pp. 107–120. DOI: 10.1007/978-3-540-70545-1_12.
- [Boe+18] Hans-J. Boehm, Olivier Giroux, Viktor Vafeiades, with input from Will Deacon, Doug Lea, Daniel Lustig, Paul McKenney, et al. *P0668R4: Revising the C++ memory model*. 2018. URL: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0668r4.html> (visited on Apr. 29, 2020).
- [BR20] Zuzana Baranová and Petr Ročkal. “Compiling C and C++ Programs for Dynamic White-Box Analysis”. In: *Workshop on Practical Formal Verification for Software Dependability (AFFORD 2019)*. Cham: Springer International Publishing, 2020. URL: <https://divine.fi.muni.cz/2019/divcc/>.
- [Bra11] Aaron R. Bradley. “SAT-Based Model Checking without Unrolling”. In: *Verification, Model Checking, and Abstract Interpretation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87. ISBN: 978-3-642-18275-4. DOI: 10.1007/978-3-642-18275-4_7.

- [Bro+16] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. “T2: Temporal Property Verification”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 387–393. ISBN: 978-3-662-49674-9. DOI: 10.1007/978-3-662-49674-9_22.
- [BRŠW15] Jiří Barnat, Petr Ročkal, Vladimír Štill, and Jiří Weiser. “Fast, Dynamically-Sized Concurrent Hash Table”. In: *Model Checking Software (SPIN 2015)*. Vol. 9232. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 49–65. ISBN: 978-3-319-23403-8. DOI: 10.1007/978-3-319-23404-5_5.
- [BSS11] Jacob Burnim, Koushik Sen, and Christos Stergiou. “Sound and Complete Monitoring of Sequential Consistency for Relaxed Memory Models”. In: *TACAS*. Berlin, Heidelberg: Springer, 2011, pp. 11–25. DOI: 10.1007/978-3-642-19835-9_3.
- [Car+16] Montgomery Carter, Shaobo He, Jonathan Whitaker, Zvonimir Rakamarić, and Michael Emmi. “SMACK Software Verification Toolchain”. In: *Proceedings of the 38th International Conference on Software Engineering Companion*. ICSE ’16. Austin, Texas: Association for Computing Machinery, 2016, pp. 589–592. ISBN: 9781450342056. DOI: 10.1145/2889160.2889163.
- [Cas+17] Franck Cassez, Anthony M. Sloane, Matthew Roberts, Matthew Pigram, Pongsak Suvanpong, and Pablo Gonzalez de Aledo. “Skink: Static Analysis of Programs in LLVM Intermediate Representation”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 380–384. DOI: 10.1007/978-3-662-54580-5_27.
- [CC14] Yan Cai and W. K. Chan. “Magiclock: Scalable Detection of Potential Deadlocks in Large-Scale Multithreaded Programs”. In: *IEEE Transactions on Software Engineering* 40.3 (Mar. 2014), pp. 266–281. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2301725.
- [CCM09] G. Canet, P. Cuoq, and B. Monate. “A Value Analysis for C Programs”. In: *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. 2009, pp. 123–124. DOI: 10.1109/SCAM.2009.22.
- [CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs”. In: *USENIX Conference on Operating Systems Design and Implementation*. OSDI’08. San Diego, California: USENIX Association, 2008, pp. 209–224. URL: <http://dl.acm.org/citation.cfm?id=1855741.1855756>.
- [CDS13] Chia Yuan Cho, Vijay D’Silva, and Dawn Song. “Blitz: Compositional bounded model checking for real-world programs”. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 136–146. DOI: 10.1109/ASE.2013.6693074.
- [CEJS98] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. “Symmetry reductions in model checking”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 147–158. ISBN: 978-3-540-69339-0. DOI: 10.1007/BFb0028741.
- [CF11] Lucas Cordeiro and Bernd Fischer. “Verifying Multi-Threaded Software Using SMT-based Context-Bounded Model Checking”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE ’11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 331–340. ISBN: 9781450304450. DOI: 10.1145/1985793.1985839.
- [CG12] Alessandro Cimatti and Alberto Griggio. “Software Model Checking via IC3”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 277–293. ISBN: 978-3-642-31424-7. DOI: 10.1007/978-3-642-31424-7_23.

- [CGMT14] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. “IC3 Modulo Theories via Implicit Predicate Abstraction”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 46–61. ISBN: 978-3-642-54862-8. DOI: 10.1007/978-3-642-54862-8_4.
- [CGS13] M. Christakis, A. Gotovos, and K. Sagonas. “Systematic Testing for Detecting Concurrency Errors in Erlang Programs”. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, pp. 154–163. DOI: 10.1109/ICST.2013.50.
- [CJ19] Eti Chaudhary and Saurabh Joshi. “Pinaka: Symbolic Execution Meets Incremental Solving”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, pp. 234–238. ISBN: 978-3-030-17502-3. DOI: 10.1007/978-3-030-17502-3_20.
- [CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. “A Tool for Checking ANSI-C Programs”. In: *TACAS*. Berlin, Heidelberg: Springer, 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15.
- [CKS19] Lucas Cordeiro, Daniel Kroening, and Peter Schrammel. “JBMC: Bounded Model Checking for Java Bytecode”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, pp. 219–223. ISBN: 978-3-030-17502-3. DOI: 10.1007/978-3-030-17502-3_17.
- [CKSY05] 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*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 570–574. ISBN: 978-3-540-31980-1. DOI: 10.1007/978-3-540-31980-1_40.
- [Cla+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. “Counterexample-Guided Abstraction Refinement”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169. ISBN: 978-3-540-45047-4. DOI: 10.1007/10722167_15.
- [CLOR19] Agostino Cortesi, Henrich Lauko, Martina Olliaro, and Petr Ročkal. “String Abstraction for Model Checking of C Programs”. In: *Model Checking Software*. Cham: Springer International Publishing, 2019, pp. 74–93. ISBN: 978-3-030-30923-7. DOI: 10.1007/978-3-030-30923-7_5.
- [Cor+18] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. “JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode”. In: *Computer Aided Verification*. Cham: Springer International Publishing, 2018, pp. 183–190. ISBN: 978-3-319-96145-3. DOI: 10.1007/978-3-319-96145-3_10.
- [CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. “Termination Proofs for Systems Code”. In: *SIGPLAN Not.* 41.6 (June 2006), pp. 415–426. ISSN: 0362-1340. DOI: 10.1145/1133255.1134029.
- [CPR07] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. “Proving Thread Termination”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 320–330. ISBN: 9781595936332. DOI: 10.1145/1250734.1250771.
- [CSV18] Marek Chalupa, Jan Strejček, and Martina Vitovská. “Joint Forces for Memory Safety Checking”. In: *Model Checking Software*. Cham: Springer International Publishing, 2018, pp. 115–132. ISBN: 978-3-319-94111-0. DOI: 10.1007/978-3-319-94111-0_7.

- [CU98] Michael A. Colón and Tomás E. Uribe. “Generating finite-state abstractions of reactive systems using decision procedures”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 293–304. ISBN: 978-3-540-69339-0. DOI: 10.1007/BFb0028753.
- [DHKR11] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. “Software Verification Using k-Induction”. In: *Static Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 351–368. ISBN: 978-3-642-23702-7. DOI: 10.1007/978-3-642-23702-7_26.
- [DHL15] Daniel Dietsch, Matthias Heizmann, Vincent Langenfeld, and Andreas Podelski. “Fairness Modulo Theory: A New Approach to LTL Software Model Checking”. In: *Computer Aided Verification*. Cham: Springer International Publishing, 2015, pp. 49–66. ISBN: 978-3-319-21690-4. DOI: 10.1007/978-3-319-21690-4_4.
- [Die+20] Daniel Dietsch, Matthias Heizmann, Alexander Nutz, Claus Schätzle, and Frank Schüßle. “Ultimate Taipan with Symbolic Interpretation and Fluid Abstractions”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2020, pp. 418–422. ISBN: 978-3-030-45237-7. DOI: 10.1007/978-3-030-45237-7_32.
- [Dil96] David L. Dill. “The Murphi Verification System”. In: *CAV*. London, UK, UK: Springer-Verlag, 1996, pp. 390–393. URL: <http://dl.acm.org/citation.cfm?id=647765.735832>.
- [DIS99] Claudio Demartini, Radu Iosif, and Riccardo Sisto. “A deadlock detection tool for concurrent Java programs”. In: *Software: Practice and Experience* 29.7 (1999), pp. 577–603. DOI: 10.1002/(SICI)1097-024X(199906)29:7<577::AID-SPE246>3.0.CO;2-V.
- [DL15] Brian Demsky and Patrick Lam. “SATCheck: SAT-Directed Stateless Model Checking for SC and TSO”. In: *SIGPLAN Not.* 50.10 (Oct. 2015), pp. 20–36. ISSN: 0362-1340. DOI: 10.1145/2858965.2814297.
- [DLW15] Matthias Dangl, Stefan Löwe, and Philipp Wendler. “CPAchecker with Support for Recursive Programs and Floating-Point Arithmetic”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 423–425. ISBN: 978-3-662-46681-0. DOI: 10.1007/978-3-662-46681-0_34.
- [DM14] Egor Derevenets and Roland Meyer. “Robustness against Power is PSpace-complete”. In: *ICAPL*. Berlin, Heidelberg: Springer, 2014, pp. 158–170. DOI: 10.1007/978-3-662-43951-7_14.
- [DMVY13] Andrei Marian Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. “Predicate Abstraction for Relaxed Memory Models”. In: *SAS*. Berlin, Heidelberg: Springer, 2013, pp. 84–104. DOI: 10.1007/978-3-642-38856-9_7.
- [DPV13] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. “Byte-Precise Verification of Low-Level List Manipulation”. In: *Static Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 215–237. ISBN: 978-3-642-38856-9. DOI: 10.1007/978-3-642-38856-9_13.
- [EMB11] Niklas Een, Alan Mishchenko, and Robert Brayton. “Efficient Implementation of Property Directed Reachability”. In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. FMCAD ’11. Austin, Texas: FMCAD Inc, 2011, pp. 125–134. ISBN: 9780983567813. URL: <https://ieeexplore.ieee.org/abstract/document/6148886/>.
- [FG05] Cormac Flanagan and Patrice Godefroid. “Dynamic Partial-order Reduction for Model Checking Software”. In: *POPL*. Long Beach, California, USA: ACM, 2005, pp. 110–121. DOI: 10.1145/1040305.1040315.

- [FLP17] Aymeric Fromherz, Kasper S. Luckow, and Corina S. Păsăreanu. “Symbolic Arrays in Symbolic PathFinder”. In: *SIGSOFT Softw. Eng. Notes* 41.6 (Jan. 2017), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/3011286.3011296.
- [Flu+16] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. “Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA”. In: *POPL*. St. Petersburg, FL, USA: ACM, 2016, pp. 608–621. DOI: 10.1145/2837614.2837615.
- [Flu+17] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. “Mixed-Size Concurrency: ARM, POWER, C/C++11, and SC”. In: *SIGPLAN Not.* 52.1 (Jan. 2017), pp. 429–442. ISSN: 0362-1340. DOI: 10.1145/3093333.3009839.
- [FMS13] Stephan Falke, Florian Merz, and Carsten Sinz. “The bounded model checker LLBMC”. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2013, pp. 706–709. DOI: 10.1109/ASE.2013.6693138.
- [Gad+18] 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 2018. Montpellier, France: Association for Computing Machinery, 2018, pp. 888–891. ISBN: 9781450359375. DOI: 10.1145/3238147.3240481.
- [Gav+19] Natalia Gavrilenko, Hernán Ponce-de-León, Florian Furbach, Keijo Heljanko, and Roland Meyer. “BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings”. In: *Computer Aided Verification*. Cham: Springer International Publishing, 2019, pp. 355–365. ISBN: 978-3-030-25540-4. DOI: 10.1007/978-3-030-25540-4_19.
- [GBHR20] Jack J. Garzella, Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. “Leveraging Compiler Intermediate Representation for Multi- and Cross-Language Verification”. In: *Verification, Model Checking, and Abstract Interpretation*. Cham: Springer International Publishing, 2020, pp. 90–111. ISBN: 978-3-030-39322-9. DOI: 10.1007/978-3-030-39322-9_5.
- [GG08] Malay K. Ganai and Aarti Gupta. “Efficient Modeling of Concurrent Systems in BMC”. In: *Model Checking Software*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 114–133. ISBN: 978-3-540-85114-1. DOI: 10.1007/978-3-540-85114-1_10.
- [GIC17] Mikhail YR Gadelha, Hussama I Ismail, and Lucas C Cordeiro. “Handling loops in bounded model checking of C programs via k-induction”. In: *International Journal on Software Tools for Technology Transfer* 19.1 (2017), pp. 97–114. DOI: 10.1007/s10009-015-0407-9.
- [Gie+17] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. “Analyzing Program Termination and Complexity Automatically with AProVE”. In: *Journal of Automated Reasoning* 58.1 (Jan. 2017), pp. 3–31. ISSN: 1573-0670. DOI: 10.1007/s10817-016-9388-y.
- [GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: Directed Automated Random Testing”. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’05. Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 213–223. ISBN: 1595930566. DOI: 10.1145/1065010.1065036. URL: <https://doi.org/10.1145/1065010.1065036>.

- [GLSW17] Henning Günther, Alfons Laarman, Ana Sokolova, and Georg Weissenbacher. “Dynamic Reductions for Model Checking Concurrent Software”. In: *Verification, Model Checking, and Abstract Interpretation*. Cham: Springer International Publishing, 2017, pp. 246–265. ISBN: 978-3-319-52234-0. DOI: 10.1007/978-3-319-52234-0_14.
- [GLW16] Henning Günther, Alfons Laarman, and Georg Weissenbacher. “Vienna Verification Tool: IC3 for Parallel Software”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 954–957. DOI: 10.1007/978-3-662-49674-9_69.
- [GMCL16] Mário Garcia, Felipe Monteiro, Lucas Cordeiro, and Eddie de Lima Filho. “ESBMC^{QtOM}: A Bounded Model Checking Tool to Verify Qt Applications”. In: *Model Checking Software*. Cham: Springer International Publishing, 2016, pp. 97–103. ISBN: 978-3-319-32582-8. DOI: 10.1007/978-3-319-32582-8_6.
- [GMCN19] Mikhail R. Gadelha, Felipe Monteiro, Lucas Cordeiro, and Denis Nicole. “ESBMC v6.0: Verifying C Programs Using k-Induction and Invariant Inference”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, pp. 209–213. ISBN: 978-3-030-17502-3. DOI: 10.1007/978-3-030-17502-3_15.
- [God+96] Patrice Godefroid, Jan van Leeuwen, Juris Hartmanis, Gerhard Goos, and Pierre Wolper. *Partial-Order Methods for the Verification of Concurrent Systems. An Approach to the State-Explosion Problem*. Vol. 1032. Springer Heidelberg, 1996. ISBN: 978-3-540-60761-8. DOI: 10.1007/3-540-60761-7.
- [God05] Patrice Godefroid. “Software model checking: The VeriSoft approach”. In: *Formal Methods in System Design* 26.2 (2005), pp. 77–101. DOI: 10.1007/s10703-005-1489-x.
- [God97] Patrice Godefroid. “Model Checking for Programming Languages Using VeriSoft”. In: *POPL*. Paris, France: ACM, 1997, pp. 174–186. DOI: 10.1145/263699.263717.
- [GPR11] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. “Threader: A Constraint-Based Verifier for Multi-threaded Programs”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 412–417. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_32.
- [Gra+15] Kathryn E. Gray, Gabriel Kerneis, Dominic Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. “An Integrated Concurrency and Core-ISA Architectural Envelope Definition, and Test Oracle, for IBM POWER Multiprocessors”. In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: Association for Computing Machinery, 2015, pp. 635–646. ISBN: 9781450340342. DOI: 10.1145/2830772.2830775.
- [Gre+17] Marius Greitschus, Daniel Dietsch, Matthias Heizmann, Alexander Nutz, Claus Schätzle, Christian Schilling, Frank Schüssele, and Andreas Podelski. “Ultimate Taipan: Trace Abstraction and Abstract Interpretation”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 399–403. ISBN: 978-3-662-54580-5. DOI: 10.1007/978-3-662-54580-5_31.
- [GW14] Henning Günther and Georg Weissenbacher. “Incremental Bounded Software Model Checking”. In: *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*. SPIN 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 40–47. ISBN: 9781450324526. DOI: 10.1145/2632362.2632374.

- [Hei+17] Matthias Heizmann, Yu-Wen Chen, Daniel Dietsch, Marius Greitschus, Alexander Nutz, Betim Musa, Claus Schätzle, Christian Schilling, Frank Schüssele, and Andreas Podelski. “Ultimate Automizer with an On-Demand Construction of Floyd-Hoare Automata”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 394–398. ISBN: 978-3-662-54580-5. DOI: 10.1007/978-3-662-54580-5_30.
- [Hen+17] Jera Hensel, Frank Emrich, Florian Frohn, Thomas Ströder, and Jürgen Giesl. “AProVE: Proving and Disproving Termination of Memory-Manipulating C Programs”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 350–354. ISBN: 978-3-662-54580-5. DOI: 10.1007/978-3-662-54580-5_21.
- [HH16] Shiyong Huang and Jeff Huang. “Maximal Causality Reduction for TSO and PSO”. In: *SIGPLAN Not.* 51.10 (Oct. 2016), pp. 447–461. ISSN: 0362-1340. DOI: 10.1145/3022671.2984025.
- [HHP13] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. “Software Model Checking for People Who Love Automata”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 36–52. ISBN: 978-3-642-39799-8. DOI: 10.1007/978-3-642-39799-8_2.
- [Hol04] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*. Vol. 1003. Addison-Wesley Reading, 2004. ISBN: 978-0-321-22862-8. URL: <https://www.cin.ufpe.br/~acm/esd/intranet/spinPrimer.pdf>.
- [Hol97] Gerard J. Holzmann. “The model checker SPIN”. In: *IEEE Transactions on Software Engineering* 23.5 (May 1997), pp. 279–295. ISSN: 0098-5589. DOI: 10.1109/32.588521.
- [Hol98] Gerard J Holzmann. “An analysis of bitstate hashing”. In: *Formal methods in system design* 13.3 (1998), pp. 289–307. DOI: 10.1023/A:1008696026254.
- [HR06] Thuan Quang Huynh and Abhik Roychoudhury. “A Memory Model Sensitive Checker for C#”. In: *FM*. Berlin, Heidelberg: Springer, 2006, pp. 476–491. DOI: 10.1007/11813040_32.
- [Hua15] Jeff Huang. “Stateless Model Checking Concurrent Programs with Maximal Causality Reduction”. In: *SIGPLAN Not.* 50.6 (June 2015), pp. 165–174. ISSN: 0362-1340. DOI: 10.1145/2813885.2737975.
- [HW07] Moritz Hammer and Michael Weber. “To Store or Not To Store” Reloaded: Reclaiming Memory on Demand”. In: *Formal Methods: Applications and Technology*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 51–66. ISBN: 978-3-540-70952-7. DOI: 10.1007/978-3-540-70952-7_4.
- [Cha+05] Sagar Chaki, Edmund Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. “Concurrent software verification with states, events, and deadlocks”. In: *Formal Aspects of Computing* 17.4 (Dec. 2005), pp. 461–483. ISSN: 1433-299X. DOI: 10.1007/s00165-005-0071-z.
- [Cha+17] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. “Data-Centric Dynamic Partial Order Reduction”. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158119.
- [Cha+20] Marek Chalupa, Tomáš Jašek, Lukáš Tomovič, Martin Hruška, Veronika Šoková, Paulína Ayaziová, Jan Strejček, and Tomáš Vojnar. “Symbiotic 7: Integration of Predator and More”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2020, pp. 413–417. ISBN: 978-3-030-45237-7. DOI: 10.1007/978-3-030-45237-7_31.

- [CHJW17] Mike Czech, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim. “Predicting Rankings of Software Verification Tools”. In: *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics*. SWAN 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 23–26. ISBN: 9781450351577. DOI: 10.1145/3121257.3121262.
- [Inv+15] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. “Lazy-CSeq: A Context-Bounded Model Checking Tool for Multi-threaded C-Programs”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015, pp. 807–812. DOI: 10.1109/ASE.2015.108.
- [ISO10] ISO C++ Standards Committee. *C++ International Standard – N3092*. 2010. URL: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf> (visited on May 2, 2020).
- [ISO11] ISO C Standards Committee. *ISO/IEC 9899:201x Committee Draft N1570*. Tech. rep. 2011. URL: <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>.
- [ISO12] ISO C++ Standards Committee. *Standard for Programming Language C++. Working Draft N3337*. Tech. rep. ISO IEC JTC1/SC22/WG21, 2012. URL: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf>.
- [ISO20] ISO C++ Standards Committee. *C++ Draft International Standard – N4860*. Mar. 31, 2020. URL: <https://isocpp.org/files/papers/N4860.pdf> (visited on May 2, 2020).
- [IT20] Omar Inverso and Catia Trubiani. “Parallel and Distributed Bounded Model Checking of Multi-Threaded Programs”. In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’20. San Diego, California: Association for Computing Machinery, 2020, pp. 202–216. ISBN: 9781450368186. DOI: 10.1145/3332466.3374529.
- [Iva+05] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. “F-Soft: Software Verification Platform”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 301–306. ISBN: 978-3-540-31686-2. DOI: 10.1007/11513988_31.
- [Jon83] C. B. Jones. “Tentative Steps toward a Development Method for Interfering Programs”. In: *ACM Trans. Program. Lang. Syst.* 5.4 (Oct. 1983), pp. 596–619. ISSN: 0164-0925. DOI: 10.1145/69575.69577.
- [Kan+15] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. “LTSmin: High-Performance Language-Independent Model Checking”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 692–707. ISBN: 978-3-662-46681-0. DOI: 10.1007/978-3-662-46681-0_61.
- [KF14] Andrey Kupriyanov and Bernd Finkbeiner. “Causal Termination of Multi-threaded Programs”. In: *Computer Aided Verification*. Cham: Springer International Publishing, 2014, pp. 814–830. ISBN: 978-3-319-08867-9. DOI: 10.1007/978-3-319-08867-9_54.
- [Kin76] James C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252.
- [KLSV17] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. “Effective Stateless Model Checking for C/C++ Concurrency”. In: *Proceedings of the ACM on Programming Languages* 2 (Dec. 2017), 17:1–17:32. ISSN: 2475-1421. DOI: 10.1145/3158105.

- [Koz82] Dexter Kozen. “Results on the propositional μ -calculus”. In: *Automata, Languages and Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 348–359. ISBN: 978-3-540-39308-5. DOI: 10.1007/BFb0012782.
- [KPV03] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. “Generalized Symbolic Execution for Model Checking and Testing”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 553–568. ISBN: 978-3-540-36577-8. DOI: 10.1007/3-540-36577-X_40.
- [KRB17] Katarína Kejstová, Petr Ročkal, and Jiří Barnat. “From Model Checking to Runtime Verification and Back”. In: *Runtime Verification*. Cham: Springer International Publishing, 2017, pp. 225–240. ISBN: 978-3-319-67531-2. DOI: 10.1007/978-3-319-67531-2_14.
- [KRS19] Temesghen Kahsai, Philipp Rümmer, and Martin Schäf. “JayHorn: A Java Model Checker”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, pp. 214–218. ISBN: 978-3-030-17502-3. DOI: 10.1007/978-3-030-17502-3_16.
- [KSH13] Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. “LCT: A Parallel Distributed Testing Tool for Multithreaded Java Programs”. In: *Electronic Notes in Theoretical Computer Science* 296 (2013). Proceedings the Sixth International Workshop on the Practical Application of Stochastic Modelling (PASM) and the Eleventh International Workshop on Parallel and Distributed Methods in Verification (PDMC)., pp. 253–259. ISSN: 1571-0661. DOI: <https://doi.org/10.1016/j.entcs.2013.09.002>. URL: <http://www.sciencedirect.com/science/article/pii/S1571066113000480>.
- [KT14] Daniel Kroening and Michael Tautschnig. “CBMC – C Bounded Model Checker”. In: *TACAS*. Berlin, Heidelberg: Springer, 2014, pp. 389–391. DOI: 10.1007/978-3-642-54862-8_26.
- [KT19] Kareem Khazem and Michael Tautschnig. “CBMC Path: A Symbolic Execution Retrofit of the C Bounded Model Checker”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, pp. 199–203. ISBN: 978-3-030-17502-3. DOI: 10.1007/978-3-030-17502-3_13.
- [Kur95] Robert P. Kurshan. *Computer-aided Verification of Coordinating Processes: The Automata-theoretic Approach*. Vol. 302. Princeton University Press, 1995.
- [Laa19] Alfons Laarman. “Optimal compression of combinatorial state spaces”. In: *Innovations in Systems and Software Engineering* 15.3-4 (2019), pp. 235–251. DOI: 10.1007/s11334-019-00341-7.
- [Lah+17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. “Repairing Sequential Consistency in C/C++11”. In: *SIGPLAN Not.* 52.6 (June 2017), pp. 618–632. ISSN: 0362-1340. DOI: 10.1145/3140587.3062352.
- [LGR11] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. “KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 609–615. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_49.
- [Lip75] Richard J. Lipton. “Reduction: A Method of Proving Properties of Parallel Programs”. In: *Commun. ACM* 18.12 (Dec. 1975), pp. 717–721. ISSN: 0001-0782. DOI: 10.1145/361227.361234.
- [LLVM20] LLVM Project. *LLVM Language Reference Manual*. 2020. URL: <http://llvm.org/docs/LangRef.html> (visited on May 2, 2020).

- [LQL12] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. “A Solver for Reachability Modulo Theories”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 427–443. ISBN: 978-3-642-31424-7. DOI: 10.1007/978-3-642-31424-7_32.
- [LR09] Akash Lal and Thomas Reps. “Reducing concurrent analysis under a context bound to sequential analysis”. In: *Formal Methods in System Design* 35.1 (2009), pp. 73–97. DOI: 10.1007/s10703-009-0078-9.
- [LRB18] Henrich Lauko, Petr Ročkai, and Jiří Barnat. “Symbolic Computation via Program Transformation”. In: *Theoretical Aspects of Computing – ICTAC 2018*. Cham: Springer International Publishing, 2018, pp. 313–332. ISBN: 978-3-030-02508-3. DOI: 10.1007/978-3-030-02508-3_17.
- [LŠRB19] Henrich Lauko, Vladimír Štill, Petr Ročkai, and Jiří Barnat. “Extending DIVINE with Symbolic Verification Using SMT”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, pp. 204–208. ISBN: 978-3-030-17502-3. DOI: 10.1007/978-3-030-17502-3_14.
- [Luc+16] Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. “JDart: A Dynamic Symbolic Analysis Framework”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 442–459. ISBN: 978-3-662-49674-9. DOI: 10.1007/978-3-662-49674-9_26.
- [LW10] Alexander Linden and Pierre Wolper. “An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models”. In: *SPIN*. Berlin, Heidelberg: Springer, 2010, pp. 212–226. DOI: 10.1007/978-3-642-16164-3_16.
- [Mad+12] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. “An Axiomatic Memory Model for POWER Multiprocessors”. In: *CAV*. Berlin, Heidelberg: Springer, 2012, pp. 495–512. DOI: 10.1007/978-3-642-31424-7_36.
- [Maz87] Antoni Mazurkiewicz. “Trace theory”. In: *Petri Nets: Applications and Relationships to Other Models of Concurrency*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 278–324. ISBN: 978-3-540-47926-0. DOI: 10.1007/3-540-17906-2_30.
- [MLB16] Jan Mrázek, Petr Bauch, Henrich Lauko, and Jiří Barnat. “SymDIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration”. In: *Model Checking Software: 23rd International Symposium, SPIN*. Cham: Springer International Publishing, 2016, pp. 208–213. ISBN: 978-3-319-32582-8. DOI: 10.1007/978-3-319-32582-8_14.
- [McK10] Paul E McKenney. “Memory barriers: a hardware view for software hackers”. In: *Linux Technology Center, IBM Beaverton* (2010). URL: <http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.06.14a.pdf>.
- [MFS12] 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, Experiments*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 146–161. ISBN: 978-3-642-27705-4. DOI: 10.1007/978-3-642-27705-4_12.
- [MH20] Malte Mues and Falk Howar. “JDart: Dynamic Symbolic Execution for Java Bytecode (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2020, pp. 398–402. ISBN: 978-3-030-45237-7. DOI: 10.1007/978-3-030-45237-7_28.

- [MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. “The Java Memory Model”. In: *Principles of Programming Languages*. POPL '05. Long Beach, California, USA: ACM, 2005, pp. 378–391. DOI: 10.1145/1040305.1040336.
- [Mrá+17] Jan Mrázek, Martin Jonáš, Vladimír Štill, Henrich Lauko, and Jiří Barnat. “Optimizing and Caching SMT Queries in SymDIVINE”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2017, pp. 390–393. ISBN: 978-3-662-54580-5. DOI: 10.1007/978-3-662-54580-5_29.
- [MS07] Rupak Majumdar and Koushik Sen. “Hybrid Concolic Testing”. In: *29th International Conference on Software Engineering (ICSE'07)*. 2007, pp. 416–426. DOI: 10.1109/ICSE.2007.41.
- [Mus+08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. “Finding and Reproducing Heisenbugs in Concurrent Programs.” In: *OSDI*. Vol. 8. 2008, pp. 267–280. URL: https://www.usenix.org/legacy/event/osdi08/tech/full_papers/musuvathi/musuvathi.pdf.
- [ND16] Brian Norris and Brian Demsky. “A Practical Approach for Model Checking C/C++11 Code”. In: *ACM Trans. Program. Lang. Syst.* 38.3 (May 2016). ISSN: 0164-0925. DOI: 10.1145/2806886. URL: <https://doi.org/10.1145/2806886>.
- [NDMP15] Alexander Nutz, Daniel Dietsch, Mostafa Mahmoud Mohamed, and Andreas Podelski. “ULTIMATE KOJAK with Memory Safety Checks”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 458–460. ISBN: 978-3-662-46681-0. DOI: 10.1007/978-3-662-46681-0_44.
- [Ngu+17] Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. “Lazy-CSeq 2.0: Combining Lazy Sequentialization with Abstract Interpretation”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 375–379. ISBN: 978-3-662-54580-5. DOI: 10.1007/978-3-662-54580-5_26.
- [Nol+19] Yannic Noller, Corina S. Păsăreanu, Aymeric Fromherz, Xuan-Bach D. Le, and Willem Visser. “Symbolic Pathfinder for SV-COMP”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, pp. 239–243. ISBN: 978-3-030-17502-3. DOI: 10.1007/978-3-030-17502-3_21.
- [Nor97] James R. Norris. *Markov Chains*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1997. DOI: 10.1017/CB09780511810633.
- [NY16] Nicholas Ng and Nobuko Yoshida. “Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis”. In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. Barcelona, Spain: ACM, 2016, pp. 174–184. ISBN: 978-1-4503-4241-4. DOI: 10.1145/2892208.2892232.
- [OD17] Peizhao Ou and Brian Demsky. “Checking Concurrent Data Structures Under the C/C++11 Memory Model”. In: *SIGPLAN* 52.8 (Jan. 2017), pp. 45–59. ISSN: 0362-1340. DOI: 10.1145/3155284.3018749.
- [OG76] Susan Owicki and David Gries. “An axiomatic proof technique for parallel programs I”. In: *Acta informatica* 6.4 (1976), pp. 319–340. DOI: 10.1007/BF00268134.
- [Päs+13] Corina S Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehrlitz, and Neha Rungta. “Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis”. In: *Automated Software Engineering* 20.3 (2013), pp. 391–425. DOI: 10.1007/s10515-013-0122-2.

- [PD95] Seungjoon Park and David L. Dill. “An Executable Specification, Analyzer and Verifier for RMO (Relaxed Memory Order)”. In: *SPAA*. Santa Barbara, California, USA: ACM, 1995, pp. 34–41. DOI: 10.1145/215399.215413.
- [Pel93] Doron Peled. “All from one, one for all: on model checking using representatives”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 409–423. ISBN: 978-3-540-47787-7. DOI: 10.1007/3-540-56922-7_34.
- [PFHM20] Hernán Ponce-de-León, Florian Furbach, Keijo Heljanko, and Roland Meyer. “Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2020, pp. 378–382. ISBN: 978-3-030-45237-7. DOI: 10.1007/978-3-030-45237-7_24.
- [PR12] Corneliu Popeea and Andrey Rybalchenko. “Compositional Termination Proofs for Multi-threaded Programs”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 237–251. ISBN: 978-3-642-28756-5. DOI: 10.1007/978-3-642-28756-5_17.
- [Pra+11] Prakash Prabhu, Naoto Maeda, Gogul Balakrishnan, Franjo Ivančić, and Aarti Gupta. “Interprocedural Exception Analysis for C++”. In: *ECOOP 2011 – Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 583–608. ISBN: 978-3-642-22655-7. DOI: 10.1007/978-3-642-22655-7_27.
- [PŠV20] Petr Peringer, Veronika Šoková, and Tomáš Vojnar. “PredatorHP Revamped (Not Only) for Interval-Sized Memory Regions and Memory Reallocation (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2020, pp. 408–412. ISBN: 978-3-030-45237-7. DOI: 10.1007/978-3-030-45237-7_30.
- [Pul+17] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. “Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8”. In: *Proceedings of the ACM on Programming Languages* 2 (Dec. 2017), 19:1–19:29. ISSN: 2475-1421. DOI: 10.1145/3158107.
- [Pul+19] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. “Promising-ARM/RISC-V: A Simpler and Faster Operational Concurrency Model”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1–15. ISBN: 9781450367127. DOI: 10.1145/3314221.3314624.
- [QW04] Shaz Qadeer and Dinghao Wu. “KISS: Keep It Simple and Sequential”. In: *SIGPLAN Not.* 39.6 (June 2004), pp. 14–24. ISSN: 0362-1340. DOI: 10.1145/996893.996845.
- [Ram+13] Mikhail Ramalho, Mauro Freitas, Felipe Sousa, Hendrio Marques, Lucas Cordeiro, and Bernd Fischer. “SMT-Based Bounded Model Checking of C++ Programs”. In: *2013 20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*. 2013, pp. 147–156. DOI: 10.1109/ECBS.2013.15.
- [RBB13] Petr Ročkal, Jiří Barnat, and Luboš Brim. “Improved State Space Reductions for LTL Model Checking of C & C++ Programs”. In: *NASA Formal Methods (NFM 2013)*. Vol. 7871. LNCS. Springer Berlin Heidelberg, 2013, pp. 1–15. ISBN: 978-3-642-38088-4. DOI: 10.1007/978-3-642-38088-4_1.
- [RBB16] P. Ročkal, J. Barnat, and L. Brim. “Model Checking C++ Programs with Exceptions”. In: *Sci. Comput. Program.* 128.C (Oct. 2016), pp. 68–85. ISSN: 0167-6423. DOI: 10.1016/j.scico.2016.05.007.

- [RBC15] Herbert Rocha, Raimundo Barreto, and Lucas Cordeiro. “Memory Management Test-Case Generation of C Programs Using Bounded Model Checking”. In: *Software Engineering and Formal Methods*. Cham: Springer International Publishing, 2015, pp. 251–267. ISBN: 978-3-319-22969-0. DOI: 10.1007/978-3-319-22969-0_18.
- [RE14] Zvonimir Rakamarić and Michael Emmi. “SMACK: Decoupling Source Language Details from Verifier Implementations”. In: *Computer Aided Verification*. Cham: Springer International Publishing, 2014, pp. 106–113. DOI: 10.1007/978-3-319-08867-9_7.
- [RG05] Ishai Rabinovitz and Orna Grumberg. “Bounded Model Checking of Concurrent Programs”. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 82–97. ISBN: 978-3-540-31686-2. DOI: 10.1007/11513988_9.
- [RICB17] Herbert Rocha, Hussama Ismail, Lucas Cordeiro, and Raimundo Barreto. “Model Checking Embedded C Software Using k-Induction and Invariants”. In: *Embedded Software Verification and Debugging*. New York, NY: Springer New York, 2017, pp. 159–182. ISBN: 978-1-4614-2266-2. DOI: 10.1007/978-1-4614-2266-2_7.
- [RMCB20] Herbert Rocha, Rafael Menezes, Lucas C. Cordeiro, and Raimundo Barreto. “Map2Check: Using Symbolic Execution and Fuzzing”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2020, pp. 403–407. ISBN: 978-3-030-45237-7. DOI: 10.1007/978-3-030-45237-7_29.
- [Roc+17] William Rocha, Herbert Rocha, Hussama Ismail, Lucas Cordeiro, and Bernd Fischer. “DepthK: A k-Induction Verifier Based on Invariant Inference for C Programs”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 360–364. ISBN: 978-3-662-54580-5. DOI: 10.1007/978-3-662-54580-5_23.
- [Roč+19] Petr Ročkai, Zuzana Baranová, Jan Mrázek, Katarína Kejstová, and Jiří Barnat. “Reproducible Execution of POSIX Programs with DiOS”. In: *Software Engineering and Formal Methods*. Cham: Springer International Publishing, 2019, pp. 333–349. ISBN: 978-3-030-30446-1. DOI: 10.1007/978-3-030-30446-1_18.
- [Roč20] Peter Ročkai. *DIVINE 4*. 2020. URL: <https://divine.fi.muni.cz/> (visited on May 23, 2020).
- [RŠB15] Petr Ročkai, Vladimír Štill, and Jiří Barnat. “Techniques for Memory-Efficient Model Checking of C and C++ Code”. In: *Software Engineering and Formal Methods*. Vol. 9276. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 268–282. ISBN: 978-3-319-22968-3. DOI: 10.1007/978-3-319-22969-0_19.
- [RŠČB18] Petr Ročkai, Vladimír Štill, Ivana Černá, and Jiří Barnat. “DiVM: Model checking with LLVM and graph memory”. In: *Journal of Systems and Software* 143 (2018), pp. 1–13. DOI: 10.1016/j.jss.2018.04.026. URL: <https://divine.fi.muni.cz/2017/divm/>.
- [RW19] Cedric Richter and Heike Wehrheim. “PeSCo: Predicting Sequential Combinations of Verifiers”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, pp. 229–233. ISBN: 978-3-030-17502-3. DOI: 10.1007/978-3-030-17502-3_19.
- [SA07] Koushik Sen and Gul Agha. “A Race-Detection and Flipping Algorithm for Automated Testing of Multi-threaded Programs”. In: *Hardware and Software, Verification and Testing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 166–182. ISBN: 978-3-540-70889-6. DOI: 10.1007/978-3-540-70889-6_13.
- [Sar+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. “Understanding POWER Multiprocessors”. In: *PLDI*. San Jose, California, USA: ACM, 2011, pp. 175–186. DOI: 10.1145/1993498.1993520.

- [Sar+12] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. “Synchronising C/C++ and POWER”. In: *Programming Language Design and Implementation*. PLDI ’12. Beijing, China: ACM, 2012, pp. 311–322. DOI: 10.1145/2254064.2254102.
- [SCL15] Felipe R. M. Sousa, Lucas C. Cordeiro, and Eddie B. de Lima Filho. “Bounded model checking of C++ programs based on the Qt framework”. In: *2015 IEEE 4th Global Conference on Consumer Electronics (GCCE)*. 2015, pp. 179–180. DOI: 10.1109/GCCE.2015.7398699.
- [Sew+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. “X86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors”. In: *Communications of the ACM* 53.7 (July 2010), pp. 89–97. ISSN: 0001-0782. DOI: 10.1145/1785414.1785443.
- [Sha+20] Vaibhav Sharma, Soha Hussein, Michael W. Whalen, Stephen McCamant, and Willem Visser. “Java Ranger at SV-COMP 2020 (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2020, pp. 393–397. ISBN: 978-3-030-45237-7. DOI: 10.1007/978-3-030-45237-7_27.
- [SKH12] Olli Saarikivi, Kari Kähkönen, and Keijo Heljanko. “Improving dynamic partial order reductions for concolic testing”. In: *2012 12th International Conference on Application of Concurrency to System Design*. IEEE. 2012, pp. 132–141. DOI: 10.1109/ACSD.2012.18.
- [SMA05] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A Concolic Unit Testing Engine for C”. In: *SIGSOFT Softw. Eng. Notes* 30.5 (Sept. 2005), pp. 263–272. ISSN: 0163-5948. DOI: 10.1145/1095430.1081750.
- [SPA94] Inc. SPARC International. *The SPARC Architecture Manual. Version 9*. Englewood Cliffs, New Jersey, USA: PTR Prentice Hall, 1994. ISBN: 0-13-825001-4. URL: <http://sparc.org/wp-content/uploads/2014/01/SPARCV9.pdf.gz>.
- [SS13] Marcelo Sousa and Alper Sen. “LLVMVF: A generic approach for verification of multicore software”. In: *Journal of Electronic Testing* 29.5 (2013), pp. 635–646. DOI: 10.1007/s10836-013-5405-9.
- [SS88] Dennis Shasha and Marc Snir. “Efficient and Correct Execution of Parallel Programs That Share Memory”. In: *ACM Trans. Program. Lang. Syst.* 10.2 (Apr. 1988), pp. 282–312. ISSN: 0164-0925. DOI: 10.1145/42190.42277.
- [ŠB18] Vladimír Štill and Jiří Barnat. “Model Checking of C++ Programs Under the x86-TSO Memory Model”. In: *Formal Methods and Software Engineering*. Cham: Springer International Publishing, 2018, pp. 124–140. ISBN: 978-3-030-02450-5. DOI: 10.1007/978-3-030-02450-5_8. URL: <https://divine.fi.muni.cz/2018/x86tso>.
- [ŠB19] Vladimír Štill and Jiří Barnat. “Local Nontermination Detection for Parallel C++ Programs”. In: *Software Engineering and Formal Methods*. Cham: Springer International Publishing, 2019, pp. 373–390. ISBN: 978-3-030-30446-1. DOI: 10.1007/978-3-030-30446-1_20. URL: <https://divine.fi.muni.cz/2019/Interm>.
- [ŠRB14] Vladimír Štill, Petr Ročkai, and Jiří Barnat. “Context-Switch-Directed Verification in DIVINE”. In: *Mathematical and Engineering Methods in Computer Science*. Vol. 8934. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 135–146. ISBN: 978-3-319-14895-3. DOI: 10.1007/978-3-319-14896-0_12.

- [ŠRB16] Vladimír Štill, Petr Ročkai, and Jiří Barnat. “Weak Memory Models as LLVM-to-LLVM Transformations”. In: *Mathematical and Engineering Methods in Computer Science, Revised Selected Papers*. Vol. 9548. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 144–155. ISBN: 978-3-319-29817-7. DOI: 10.1007/978-3-319-29817-7_13.
- [ŠRB16a] Vladimír Štill, Petr Ročkai, and Jiří Barnat. “DIVINE: Explicit-State LTL Model Checker”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2016, pp. 920–922. ISBN: 978-3-662-49674-9. DOI: 10.1007/978-3-662-49674-9_60.
- [ŠRB17] Vladimír Štill, Petr Ročkai, and Jiří Barnat. “Using Off-the-Shelf Exception Support Components in C++ Verification”. In: *IEEE International Conference on Software Quality, Reliability and Security (QRS)*. July 2017, pp. 54–64. DOI: 10.1109/QRS.2017.15. URL: <https://divine.fi.muni.cz/2017/exceptions/>.
- [Šti16] Vladimír Štill. “LLVM Transformations for Model Checking”. Master’s Thesis. Masaryk University, Faculty of Informatics, Brno, 2016. URL: http://is.muni.cz/th/373979/fi_m/ (visited on Mar. 1, 2016).
- [Šti18] Vladimír Štill. *Memory-Model-Aware Analysis of Parallel Programs*. Advanced Master’s thesis, PhD Thesis Proposal. 2018. URL: <https://is.muni.cz/th/uuunu/?lang=en>.
- [Tas+12] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. “TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs”. In: *Formal Techniques for Distributed Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 219–234. ISBN: 978-3-642-30793-5. DOI: 10.1007/978-3-642-30793-5_14.
- [Tom+15] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. “Verifying Concurrent Programs by Memory Unwinding”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 551–565. ISBN: 978-3-662-46681-0. DOI: 10.1007/978-3-662-46681-0_52.
- [Tom+16] Ermenegildo Tomasco, Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. “MU-CSeq 0.4: Individual Memory Location Unwindings”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 938–941. ISBN: 978-3-662-49674-9. DOI: 10.1007/978-3-662-49674-9_65.
- [Tom+17] Ermenegildo Tomasco, Truc Lam Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. “Using Shared Memory Abstractions to Design Eager Sequentializations for Weak Memory Models”. In: *Software Engineering and Formal Methods*. Cham: Springer International Publishing, 2017, pp. 185–202. ISBN: 978-3-319-66197-1. DOI: 10.1007/978-3-319-66197-1_12.
- [TVD14] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. “GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation”. In: *OOPSLA*. Portland, Oregon, USA: ACM, 2014, pp. 691–707. DOI: 10.1145/2660193.2660243.
- [VG20] Willem Visser and Jaco Geldenhuys. “COASTAL: Combining Concolic and Fuzzing for Java (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2020, pp. 373–377. ISBN: 978-3-030-45237-7. DOI: 10.1007/978-3-030-45237-7_23.
- [Vis+03] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. “Model checking programs”. In: *Automated software engineering* 10.2 (2003), pp. 203–232. DOI: 10.1023/A:1022920129859.

- [VN13] Viktor Vafeiadis and Chinmay Narayan. “Relaxed Separation Logic: A Program Logic for C11 Concurrency”. In: *OOPSLA*. Indianapolis, Indiana, USA: ACM, 2013, pp. 867–884. DOI: 10.1145/2509136.2509532.
- [YDLW19] L. Yin, W. Dong, W. Liu, and J. Wang. “Parallel Refinement for Multi-Threaded Program Verification”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 643–653. DOI: 10.1109/ICSE.2019.00074.
- [YG04] Karen Yorav and Orna Grumberg. “Static analysis for state-space reductions preserving temporal logics”. In: *Formal Methods in System Design* 25.1 (2004), pp. 67–96. DOI: 10.1023/B:FORM.0000033963.55470.9e.
- [YGL04] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. “Memory-Model-Sensitive Data Race Analysis”. In: *ICFEM*. Berlin, Heidelberg: Springer, 2004, pp. 30–45. DOI: 10.1007/978-3-540-30482-1_11.
- [Yin+18] Liangze Yin, Wei Dong, Wanwei Liu, Yunchou Li, and Ji Wang. “YOGAR-CBMC: CBMC with Scheduling Constraint Based Abstraction Refinement”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2018, pp. 422–426. ISBN: 978-3-319-89963-3. DOI: 10.1007/978-3-319-89963-3_25.
- [ZJ08] Anna Zaks and Rajeev Joshi. “Verifying Multi-threaded C Programs with SPIN”. In: *Model Checking Software*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 325–342. ISBN: 978-3-540-85114-1. DOI: 10.1007/978-3-540-85114-1_22.
- [ZKW15] Naling Zhang, Markus Kusano, and Chao Wang. “Dynamic Partial Order Reduction for Relaxed Memory Models”. In: *PLDI*. Portland, OR, USA: ACM, 2015, pp. 250–259. DOI: 10.1145/2737924.2737956.