Faculty of Informatics
Masaryk University

Model Checking Software

PhD Thesis

Petr Ročkai

January 2015

Abstract

The systems we design are increasing in complexity and it is often beyond human capacity to verify their correctness. Research in formal methods has provided us with automated tools that can help ensure correctness of a system that is beyond our capabilities to check manually. Nowadays, for a variety of reasons, such tools are applied only to a very limited number of systems. The broad goal of this thesis is to improve applicability of model checking in design and implementation of general-purpose computer software.

This thesis seeks to contribute to the state of the art in a number of areas closely aligned with this goal. Primarily, we describe and implement an interpreter for LLVM bitcode – a centrepiece of a software model checker. We then introduce a number of improvements over the basic interpreter: τ and $\tau+$ reduction to trim state spaces of threaded programs, heap symmetry reduction to make model checking with dynamic memory practical, exception handling which allows for verification of realistic C++ code. In order to successfully confront the realities of complex software behaviour, we propose a language for describing global program state. Both standalone and in combination with LTL, it aims to be succinct, precise and natural. To further leverage the unique features available in our explicit-state engine, we suggest a scheme for abstracting and refining LLVM bitcode in a composable manner.

Finally, a vital ingredient in a successful verification tool is the ability to expediently deal with realistic problems, both in terms of size and complexity. A modern implementation clearly needs to take advantage of parallel processors: we explore what it takes to do so and press the state of the art forward in certain areas, obtaining results relevant in the wider context of general-purpose data structures.

Significant advances have been made in the areas mentioned, even though many issues are still open and a subject of ongoing research. Most importantly, our key contributions lead to a practical implementation of a scalable, automated verification tool, DIVINE. We demonstrate its capabilities by using DIVINE to verify important portions of its own code base, as well as parts of the underlying standard library.

Keywords

Model checking, LLVM, Explicit-state model checking, Software model checking, Linear Temporal Logic, Parallel algorithms, Implementation, Multi-threading, Shared memory, Distributed memory, State space reductions, Partial order reduction, Interpreter, Exception handling

Acknowledgements

Many people have contributed to making this thesis possible. I would like to thank the whole Parallel and Distributed Systems Laboratory – it has been a great place to learn and work. Namely, I would like to thank to doc. RNDr. Jiří Barnat, PhD. and prof. RNDr. Luboš Brim, CSc. for jointly providing excellent guidance throughout my study and co-authoring the papers which led to this thesis. I would also like to thank prof. RNDr. Ivana Černá for her great lectures and great listening skills.

Further thanks go to Red Hat, Inc. and particularly Tom Coughlan, for making it possible to allocate part of my working hours for my research activities, directly contributing to this thesis, and to JCMM (Jihomoravské centrum pro mezinárodní mobilitu) which has provided me with a 3-year scholarship.

Last, but not least, I would like to thank my family, my friends and all the good people who helped me in keeping my sanity and focus, and made the time generally enjoyable. Finally, I would like to thank L., with whom came the music.

Allons-y!

Contents

1 Introduction 7
1.1 Model Checking 8
1.2 Parallel Programs 8
1.3 Model Checking Software 9
1.4 DIVINE 10
1.5 Organisation of This Thesis 11
1.6 Conventions 13

2 State of the Art 15
2.1 Model Checking 15
2.2 Model Checking Software 17
2.3 Program Semantics 21
2.4 Symbolic Methods 25
2.5 Explicit Methods & Scalability 29
2.6 Algorithms for Accepting Cycle Detection 31
2.7 State Space Reductions 42
2.8 Linear Temporal Logic 47

3 Parallel Search Implementation 51
3.1 Data Structure & Algorithm Design 52
3.2 IPC Queues 55
3.3 Termination Detection 61
3.4 Shared Queues 64
3.5 Hash Tables 66
3.6 Compression 75
3.7 Memory Allocation 77

4 LLVM 83
4.1 Language & Bitcode 83
4.2 Semantics 91
4.3 Control Flow 91
4.4 Heap 92
4.5 Implementation 94
4.6 Exception Handling 101
4.7 Counterexamples 108

5 Property Specification 111
5.1 Safety 111
5.2 LTL 114
5.3 Atomic Propositions 115

6 Reductions . . . 121
6.1 Partial Order Reduction . . . 121
6.2 $\tau-$ and $\tau+$reduction . . . 128
6.3 Heap Symmetry Reduction . . . 132

7 Abstraction & Refinement . . . 135
7.1 LART . . . 135
7.2 Predicate Abstraction . . . 136
7.3 Counterexample Analysis . . . 137
7.4 Refinement . . . 138
7.5 Implementation . . . 138
7.6 Alias Analysis . . . 140
7.7 Per-Value Abstraction . . . 143
7.8 Relational Abstractions . . . 147

8 Conclusions . . . 151
8.1 Contribution . . . 151
8.2 Future Work . . . 152

A C++ Bricks . . . 155
A.1 Heterogeneous Lists . . . 156
A.2 Remote Procedure Calls . . . 156
A.3 Tagged Unions and Algebraic Data . . . 157
A.4 Recursive-Descent Parsing . . . 158
A.5 Bit-Level Operations . . . 159
A.6 Command Line Parsing . . . 160
A.7 Unit Testing . . . 160
A.8 Functional Testing . . . 162
A.9 Benchmarks . . . 162
A.10 Others . . . 163

B Measurement Data . . . 165
B.1 IPC Queues . . . 165
B.2 Shared Queues . . . 167
B.3 Hash Tables . . . 168

C Related Papers . . . 177

References . . . 179

1 Introduction

The main concern of formal methods in general, and model checking in particular, is helping to design "correct" systems. Informally, a correct system is one that does the right thing (i.e. performs the tasks it has been designed to perform) and does it right (i.e. the result of the performed task is the desired one). Nevertheless, it is difficult to capture these notions even informally – formal specification is still more difficult. Moreover, to formally reason about any system, the system itself needs to be described in a rigorous, formal fashion.

Usually, when applying formal methods, the correctness requirements are expressed as a set of properties, laid down in a suitable logic, that the system must satisfy to be considered correct. This formal description usually originates in a less formal, natural-language description of safety and behaviour requirements for the device or system in question. It is (a very important) part of the work of the human verifier to ensure that the informal notion of correctness corresponds to the formal properties used in formal (whether manual or automated) verification.

In contrast, testing is a lightweight method that dispenses with the need of formal specification and chooses a different set of trade-offs, compared to formal methods. Testing favours simple procedures, which require less expertise and are cheap to implement. On the other hand, when using testing alone, a small increase in certainty requires large (exponential) increase in testing coverage (and hand-in-hand, cost). At low to medium assurance levels, this exponential increase is offset by the very low cost of a given fixed amount of testing. However, over a certain threshold, the exponential character of the cost behaviour necessarily dominates.

On the other hand, the cost of applying formal methods has an entirely different cost characteristic [31]. While the initial investment is high (and therefore, the cost curve very steep, closely trailing the steep *learning curve* of formal methods), after a certain level, it flattens out and grows quite slowly.

In many applications, testing provides a very reasonable trade-off: especially in the light of cost associated to deployment of formal methods. Moreover, the existing tool support for formal methods is still mostly academic, and adoption of formal methods in industry is lagging behind testing-based approaches to correctness evaluation.

Despite these issues, the acceptance of formal methods is slowly advancing and they are increasingly relied upon for real-world systems. For a recent example, let us just mention [102], a report about replacement of testing with symbolic model checking in the Intel Core i7 processor design. Moreover, explicit-state model checking of computer programs is known to have been exploited by some high-profile producers of

critical software components (NASA JPL, the birthplace of the venerable SPIN model checker [87], among others).

1.1 Model Checking

In full generality, the term model checking describes an automated process of verifying the fact that a given structure (a model) satisfies a given logical formula. The formula needs to be specified in a suitable logic: this could be as simple as propositional logic, although most often, some kind of temporal logic [64] is used in conjunction with model checking in practice. Temporal logics allow the user to describe behaviour of a system in time – which allows useful statements to be made about hardware or software systems. Indeed, proving (or disproving) properties of software or hardware systems is currently the most common application of model checking.
Of course, there are certain limitations that need to be imposed on the models for the model checking to be practically useful. It is required that the model is finite, as to be fully constructible by a computer: this may be especially problematic in case of software. Nevertheless, there are important classes of software systems that are finite, and therefore where fully automatic model checking is applicable. Moreover, there are even approaches for model-checking infinite-state systems, but these are out of the scope of the proposed thesis – we will only discuss finite models in the following.
Even more specifically, we will concern ourselves with model checking of properties given as Linear Temporal Logic (or LTL, for short) formulae. This is one of the temporal logics that are in current widespread use in model checking: the other being CTL – Computation Tree Logic and CTL* and various subsets of either (in fact, LTL is itself a subset of the latter). In some cases, special consideration will be given to properties that can be checked by a simpler algorithm (reachability) than required for full LTL model checking. In general, safety conditions like absence of deadlocks and absence of assertion violations fall into this category and are both very attractive targets for automated, exhaustive checking.

1.2 Parallel Programs

Due to hardware development, parallel programming has become a very important topic of research. In the current state of the practice, the overwhelming majority of parallel programs are lock-based: access to shared resources is controlled by acquisition and release of exclusive locks. The major advantage of this approach is efficiency: well-written lock-based parallel programs can achieve very good scalability, with very little sequential overhead. The major disadvantage, then, is lack of composability. In fact, locking is inherently non-composable, a fact that makes testing of parallel programs a very gruesome matter.
The practical consequences of non-composability are twofold: when implementing the program, it is hard to control the non-local effects of lock acquisition, and easy to introduce races or deadlocks. Moreover, testing effort cannot be easily decomposed to

subsystems, because locking interaction of the different components needs to be tested as a whole. Therefore, in parallel programs, whole-system testing will often invalidate the positive testing results of its constituent components: much more often than it is the case in sequential designs.
Even though testing lock-based parallel programs is hard on the grounds of non-composability of locking alone, another issue compounds the problem, moving it into the domain of nearly impossible: independent threads of execution can interleave arbitrarily, which makes testing results extremely hard to reproduce. Together, these two problems present an insurmountable barrier to successful testing of parallel programs.
There are basically two options to deal with this problem. The first is to avoid lock-based parallelism completely, using a different parallel programming paradigm: the main candidates in this area are data parallelism [83] and (software) transactional memory [80, 82 and 139]. While data parallelism (as opposed to control parallelism) is much more robust, its applicability is limited to a certain class of problems. On the other hand, software transactional memory offers a composable approach to control parallelism, which in itself is a very promising concept. Unfortunately, an efficient, practically applicable implementation of software transactional memory has proven to be quite elusive.
The other option in the locking dilemma is to find a suitable substitute for testing that is more suitable to tackle the problem at hand. The one major candidate is explicit-state model checking, although it comes with its own set of limitations and problems. Nevertheless, combined with the high assurance potential of model checking in general, we believe it can become a viable solution in many cases. Moreover, the opportunity to apply model checking directly to programs, at a very low level (post-compilation and, equally importantly, post-optimisation), uncovers new potential for the technique, while at the same time posing new challenges.

1.3 Model Checking Software

In this thesis, we explore the latter of these two approaches: applying model checking to parallel programs, in order to verify their correctness, as specified by a set of LTL properties. We will focus on programs that exhibit control-level parallelism: this is an area where it has the highest potential value and where few real alternatives exist. Nevertheless, model checking can be applied more generally to other types of programs, and we do not preclude any such applications.
A traditional hurdle in applying model checking is the requirement to maintain a separate model of the system, in addition to the actual implementation. While using a separate modelling language is advantageous in the early stages of system specification and design, this advantage is reversed later in the process of implementation. It is an extremely rare occurrence when a design is fixed at some point before implementation starts and is never revised. In a vast majority of cases, the design will co-evolve as the implementation progresses: it is in those cases that the requirement to maintain an accurate model, completely separate from the implementation, becomes a major

hindrance. On the other hand, if model checking could be applied directly to the implementation, this problem would be largely mitigated.
In addition to the extraneous maintenance burden, keeping the model and the implementation separate has another major disadvantage: while design-level bugs represent a major class of software defects, when control-level parallelism is taken into account, implementation bugs comprise a class equally important. Naturally, these implementation-level bugs cannot be uncovered through model checking the design alone.
Therefore, to address these two concerns, it is desirable to apply model checking to the program directly. Moreover, to further improve the fidelity of the model checking process, it is advantageous to apply the model checker as close to the final executable form of the program as is feasible. This means that the input to the model checker should have already been through all the code transformations that are routinely done by compilers as part of code generation and optimisation. While model checking machine code may be an option, there are nowadays multiple machine-independent, assembly-level languages that may offer better trade-offs than machine code would.
Of course, applying model checking to programs directly is not devoid of challenges. The most notorious problem of explicit-state model checking of parallel systems is the so-called "state space explosion", pertaining to the fact that the size of the state space is exponential in the number of parallel processes. It is important to say that this problem is *inherent* and cannot, in principle, be "solved", as it is directly tied to the interleaving behaviour of parallel programs. However, this does not mean it cannot be fought, and for practical purposes even defeated. Moreover, the exponential growth observed in explicit-state model checking is dwarfed by the double exponential growth that plagues testing.[1] In the next chapter, we will survey the existing techniques to fight the state space explosion problem in more detail. For now, we content ourselves with the assumption that this problem, while important, is not entirely fatal.

1.4 DIVINE

An important part of this thesis is the implementation of a software model checker, DIVINE[2]. As outlined in previous sections, model checking is still an expensive undertaking, and we lack a mature tool which could handle real-world software with a high degree of automation; DIVINE is a practical contribution toward this goal.
First versions of DIVINE emerged as an experimental vehicle for parallel distributed-memory model checking of asynchronous systems with the clear motivation to increase its power over that of sequential model checkers [7]. A modelling language, DVE [140],

[1] To thoroughly test a parallel system, one would need to test every possible execution path. Even when we only account for minimal execution paths (of which, there is a finite number in a finite-state system), the number of such minimal execution paths is exponential in the size of the state space, which is in turn exponential in the number of parallel processes. A model checker, on the other hand, only needs to construct the state space, avoiding construction of paths through that state space.

[2] DIVINE is an ongoing open-source project, and up-to-date information, source code and user-level documentation can all be found at **http://divine.fi.muni.cz**.

was created along with early versions of the tool. At that time, the primary platform for DIVINE was an MPI-connected cluster, or a network of workstations. However, a shift in the hardware market, towards SMP systems with significantly larger amount of memory, had soon become apparent. Early on, the focus in development of DIVINE moved towards efficient use of commodity hardware. The effort to improve efficiency on SMP, shared-memory architectures culminated in the release of DIVINE Multi-Core [11], based on distributed algorithms [17] adapted for better shared-memory performance. The distributed and the shared-memory branches have been eventually merged to form DIVINE 2.0 [18], with the latest stable release on this branch being DIVINE 3.2 [8].
Nowadays, DIVINE is a modern explicit-state model checker, building on high-performance algorithms and data structures, offering unparalleled versatility. It scales from a typical developer's laptop, up to a high-end compute cluster. It can verify a wide range of languages, including C and C++:

- LLVM bitcode (suitable for model checking C and C++ code)
- UPPAAL [24] timed automata
- DVE [140] – the original DIVINE modelling language
- user-implemented, compiled models via CESMI[3]
- MurPHI [56], including symmetry reduction
- CoIn [36], for modelling with component interaction automata

1.5 Organisation of This Thesis

In this thesis, we will discuss a number of topics, all of them important components in the design and implementation of a modern software model checker. As such, the techniques presented in this thesis have all been used in the implementation of DIVINE.
Chapter 2 recounts the state of the art in software model checking, explicit-state and symbolic model checking methods, state space reductions for explicit-state model checking, and the property specification logics that are in use in conjunction with model checking of software.
Chapter 3 is focused on the implementation aspects of a parallel model checker: efficient graph traversal, data structures and hardware considerations. Since model checking is a computationally intensive task, scalable parallelisation is an important aspect of every practical implementation, on par with algorithmic efficiency.
Chapter 4 is concerned with the specifics of model checking software written in common high-level programming languages, leveraging existing compilers and low-level translation of programs. We describe the LLVM language itself, how C and C++ are translated into LLVM, how to deal with language runtime libraries and system interfaces, how to approach exception handling of high-level languages like C++, and finally

[3] CESMI has been successfully used to implement a 3rd-party module for verification of Matlab Simulink designs of avionics [6].

we briefly discuss the challenges and specifics of counterexample generation in this context.
In **Chapter 5**, we argue the importance of state-space reductions, both general and those specific to model checking of low-level code. We describe the most important reductions in detail and provide efficient algorithms to compute them in the context of parallel graph traversal.
In addition to reductions (**Chapter 6**), abstractions discussed in **Chapter 7** present a powerful way to fight the state-space explosion problem. We discuss the well-known CEGAR approach and how to apply it in the context of LLVM verification. An important aspect of this endeavour is communication between the model checker proper and the abstraction engine and the representation of counterexamples. We also discuss the applicability of DIVINE's LLVM abstraction engine outside of explicit-state model checking.
Finally, **Chapter 8** wraps up the broad spectrum of topics, how each contributed to the success of DIVINE and maps out directions for possible future research and improvements.

1.5.1 Contribution

Each of the following chapters constitutes an area where this thesis contributes to the state of the knowledge in the field of formal verification (and falsification). Besides the summary in **Chapter 2**, each chapter in some sense improves on the current state of the art.
We would like to point out that the primary contribution of this thesis cannot be found, as such, in the pages you are reading: it can be instead found in the source code of the model checker DIVINE. While source code is often highly technical and subject to various language- and architecture-specific limitations, it is still the most rigorous description (even if not the most accessible) of the ideas presented herein.

Chapter 3 Parallelism We introduce, verify correctness and quantify the performance of a number of crucial building blocks: algorithms and data structures. These building blocks allow us to retain a high level of abstraction in the description of model checking algorithms in DIVINE, while at the same time offering superior performance. The building blocks also provide great potential for re-use in other endeavours, outside of model checking.

Chapter 4 LLVM We show how to build an explicit-state model checker for LLVM. We consider how to represent memory and how to treat pointers and how to deal with exception mechanisms often used in higher-level programming languages. We also extensively deal with the support code needed by real-world programs: language runtimes and system and standard libraries; and with the interface between the model checker and the program being model-checked (analogous to the interface that the operating system provides to system libraries).

Chapter 5 Properties We classify the properties commonly sought in programs and look at how these map to model checking as provided by DIVINE and especially its LLVM support. We explore the

combination of LTL properties with software (as opposed to models), a topic largely ignored in the literature. We introduce a novel approach to specifying such properties succinctly and intuitively.

Chapter 6 Reductions We describe a new reduction ($\tau+$) tailored to explicit-state model checking of LLVM bitcode with parallelism, as well as an implementation of heap symmetry reduction in an environment with unrestricted, untagged pointers. Moreover, we elaborate an improvement of the pre-existing *partial order reduction*, suitable for parallel search algorithms.

Chapter 7 Abstraction Finally, we propose a novel approach to abstraction in the context of LLVM, as a composable program transformation. This provides us with a framework to easily implement and evaluate various abstractions in a real-world context. Our approach, just as importantly, closes the gap between the way we think about abstractions and the way they are implemented in practice in model checkers.

1.6 Conventions

There are multiple areas where aspects of presentation deserve closer explanation. We cover such cases together in this section for easy reference. While typographical conventions should be rather self-explanatory, we do note that paragraphs labelled as "note" and typeset in a lighter shade of text are reserved for low-level technical information. Reading such notes is strictly optional.

1.6.1 Plots

The plots in this thesis use a ribbon style, where the central line represents the mean value and the ribbon around it represents error margins. As explained in **Section 3.1.2**, we use bootstrap to obtain confidence intervals on all measurements. Moreover, since it is impractical at best to sample the parameter space finely enough, we use quadratic spline interpolation to estimate values between data points. Actual measurements are displayed as black dots in the plots. The y axis is always scaled for best detail, please always remember to consult y markings when comparing plots to each other. We have strived, on the other hand, that the x and z axes are consistent across all the plots (over the same parameter space).

1.6.2 Algorithms

In several places in this thesis, we describe algorithms. It is common that pseudocode of the algorithm is included for reference in such cases. There is, however, a major downside to how pseudocode is traditionally used: it cannot be executed, and as a result, it is far too easy to introduce errors that go unnoticed. We have opted to instead use a very concise, but executable form for writing down algorithms. The (pseudo)code found in this thesis is written in the programming language Haskell, and is bundled

in executable form in an (electronic-only) attachment. While some readers may be unfamiliar with Haskell, we believe the small subset we use, tailored for high-level algorithm description, is sufficiently self-explaining to pose no threat to comprehension. Nevertheless, there are a few conventions worth mentioning explicitly: ⇐ operator stores a value (on the right hand side) into a mutable state variable (left hand side), while the ← syntactic shortcut binds result of a computation to a name – creating an immutable variable. The `Computation A B` type describes a computation with result `A` and global mutable state `B`. While it is customary to use set-based notation in pseudocode, we use lists whenever sets are normally used. The usual operations ($\in, \cup, \cap$ and set comprehension) work exactly as they do in sets, the only difference being the square brackets ([] vs. {}).

In the electronic (PDF) version of this thesis, each algorithm is accompanied by the symbol you can see to the left of this paragraph. This icon represents an attached file – the Literate Haskell source file, in the case of algorithms. The file attached to this paragraph is the small `Pseudocode.lhs` library that all the algorithms require to function.

note Technically, the algorithms are implemented in terms of a simple monad, built from a state monad transformer stacked on top of a continuation monad. This makes it convenient to express control flow and to keep mutable state, while at the same time making both these aspects explicit. The mutable state is implemented using records with `fclabels`; each time we define mutable state of an algorithm, an implicit call to `mkLabels` is placed after the corresponding `data` definition. It is elided in algorithm presentation as this information doesn't add any real value.

2 State of the Art

In this chapter, we will survey the best known answers to the questions and challenges outlined in the Introduction, providing a concise summary, augmented with references to more detailed review works, as well as to individual technical papers.

2.1 Model Checking

While the concept of model checking is extremely general, we will focus on a specific subclass – model checking of temporal properties. As far as temporal behaviour is concerned, the natural structure to use as the model is a Kripke structure [104], a triple $(s_0, S, \rightarrow)$ where $s_0 \in S$ and $\rightarrow \subseteq S \times S$. The set S is a set of "states" (or "worlds"): each element of this set represents an instant in the execution of the system (or in the evolution of a world). The single state s_0 represents the initial state of the system (the real "world") and $\rightarrow$ represents the transitions of the system (from one state to another).
The structure itself corresponds to the featureless branching (and possibly looping) behaviour of a system. For any practical use, we need to describe the individual states of the system in more detail. For that, we use a simple propositional calculus: we introduce a set of propositions true in a given "world", or a system state. Technically, we define L a set of all possible propositions (or labels), and $\psi : S \mapsto 2^L$ a function that describes each of the states in terms of these propositions.

2.1.1 Symbolic vs Explicit

Automated model checking is always based on some sort of a search. The state space of a system is systematically explored, ensuring that the state space as a whole satisfies the properties required.
There are two main broad approaches to model checking temporal properties, distinguished by the way system states are represented in the model checker. The simpler approach is to *enumerate* all the system states explicitly, storing each as a separate entity in memory during a search. On the other end of the spectrum lie methods which represent the entire state space as a symbolic set. The usual set representations range from BDDs (Binary Decision Diagrams) [38], through propositional formulas [27 and 45], to first order logics with theories [3]. The latter two are usually found in systems which apply model checking to a certain set of safety properties, which is a strict subset of temporal properties (in the form of the LTL formula $\mathcal{G}\phi$ for some propositional formula ϕ), while BDDs are often used in conjunction with CTL properties.
The first apparent difference is that of complexity: an explicit-state model checker is a relatively simple matter, building successor states and storing each, exploring the state space systematically. A symbolic model checker needs to perform complex symbolic

manipulation to build its representation. On the flip side, the obvious advantage for a symbolic model checker is its vastly more compact representation of the state space. Of course, reality is not nearly as simple, the line between the two worlds is not quite sharp nor straight, and the trade-offs involved are more intricate. In practice, all realistic model checkers are symbolic to some degree, whether by the virtue of a partial order reduction, symmetry reduction, or through use of abstractions. On the other hand, explicit-state control flow seeps into the more symbolically minded world in form of "concolic" (portmanteau of concrete and symbolic) methods or bounded-interleaving symbolic execution [136 and 138].

2.1.2 Automata-based Approach

Contemporary explicit-state LTL model checkers are largely based on a scheme proposed in [150]. The basic idea is to translate a negated LTL property into a Büchi automaton (which is called a negative claim automaton in this context). This automaton then accepts a language that corresponds to the runs violating the original LTL property. In itself, this automaton will accept many words – however, when a synchronous product is done with the modelled system, we obtain an automaton accepting an intersection of two languages: the first one containing all the runs in the original modelled system, and the second containing those that violate the desired LTL property. The model checking problem then reduces to verifying that the product automaton accepts exactly the empty language; moreover, if the language accepted is non-empty, the (infinite) accepted words represent the undesirable behaviours (according to the LTL property used), i.e. a set of counterexamples. Finally, the problem of language emptiness for Büchi automata is a relatively easy problem: the language is non-empty iff there is a reachable accepting cycle in the graph of the automaton.
Moreover, the model specification is usually not given as an explicit state graph of the entire system: this would be, in most cases, rather impractical. The preferred input is in the form of a set of (usually) small extended automata, augmented with communication – the full state space of the system is then constructed on the fly from this compact description. Unfortunately, the resulting size of the state space is exponential in the number of constituent processes.
Now we can identify the two most resource-demanding portions of the actual LTL model checking: the construction of the full state space from the model, and the search for reachable accepting cycles. In practice, these two processes are often interleaved – the construction of the state space is driven by the demands of the accepting cycle detection algorithm: only the parts that the cycle detection explores are computed, and only when they are needed.
Quite importantly, this approach to LTL verification is incompatible with most of the set-based symbolic methods outlined above. It is, however, possible to combine automata-based LTL model checking with symbolic methods based on abstraction/refinement.

2.2 Model Checking Software

The traditional LTL model checkers, as described in **Section 2.1.2**, rely on a description of the (parallel, asynchronous) system in the form of extended finite automata. This is usually realised through a special "modelling" language (cf. ProMeLa, DVE, μCRL, etc.). Nevertheless, applying model checking to unmodified, or only lightly annotated software systems written in general-purpose programming languages is extremely desirable. Among other advantages, this approach substantially reduces costs associated with model checking, which – as outlined in **Chapter 1** – is often an expensive undertaking. This practice is, in some literature, also called code model checking. Tools that bypass the modelling step, i.e. those, that model-check software directly, remove the need for a significant part of the specialist work required for model checking (which is how the cost savings are achieved, along with making the process more straightforward and less prone to latency). These savings in turn enable wider applicability of formal methods in general.

A number of advancements have been made in the area of software model checking. One of the first examples is the support for combining C code with ProMeLa models in SPIN, which can be used, although with a number of caveats and substantial amount of extra work, to verify implementation-level properties. Another early approach to the problem is constituted by automated model extraction [54, 84 and 93]. The ZING [2] model checker is shipped with automated model extraction tools as well.

More direct approaches, which are in many cases also easier to apply, are embodied by model checkers based on a particular programming language (or runtime), like CMC [122], JCat [58], Java PathFinder [151] and MoonWalker [57]. Nevertheless, these existing solutions to this problem are still somewhat limited in their applicability: most are tied to a specific programming language, and often even to a specific subset of that language. On the other hand, even though both Java PathFinder and MoonWalker are, for most practical purposes, tied to Java and C# respectively, they are in fact targeting JVM and .NET in general. This means that they can be used with other languages, as far as these can be hosted by the respective runtime environment. Yet another approach, pioneered by GMC [106], the GIMPLE Model Checker, is to leverage an intermediate representation of a particular compiler, in this case GCC. In this case, the downside is that GIMPLE is not supported as an external interface by GCC, and the required support infrastructure is more or less internal to GCC.

An important sub-class of these direct approaches is constituted by systems based on CEGAR ([48], CounterExample-Guided Abstraction Refinement), like BLAST [81]. Another popular technique for model checking of software is bounded model checking, which puts an adaptive upper bound on loop unrolling, ensuring finiteness of the model checker input [27]. This approach is used by, for example, CBMC [51 and 105], a tool that applies bounded model checking to C code.

The latest crop of bounded model checkers has made a similar choice as we did in DIVINE, opting to use the LLVM IR as their input formalism. Two such tools are LLBMC [71] and NBIS [79]. Both use SMT solvers as their backend.

Both CEGAR-based and bounded software model checkers primarily deal with open-ended programs, where a fully non-deterministic environment is assumed, and the job of the model checker is to ensure the program behaves correctly regardless of the arbitrary behaviour of the environment. In contrast to this, most LTL model checkers work with closed programs, that have no unspecified "outside the program" environment, and as such no input-output behaviour. From a theoretical standpoint, there is no difference: since all model checkers admit non-deterministic choice as a building block of the programs under scrutiny, any environment, including fully non-deterministic, arbitrary-behaviour one, as assumed by the open-ended tools, can be recreated as part of the program. The technical difference lies in how different tools deal with "wild" non-determinism arising from such an environment: abstraction-based and bounded symbolic tools are essentially built to deal with this type of behaviour. On the other hand, explicit-state tools can in theory handle such programs, but in practice even reading a single integer from the environment will easily overwhelm the capacity of the system. Hence, even if the distinction is theoretically meaningless, it has substantial practical consequences.
Besides this section, a fairly complete survey on the state of the art in software model checking can be found in [152].

2.2.1 LLVM

LLVM (previously also known as the Low-Level Virtual Machine) [113] has been primarily developed as a program compilation framework (both ahead of time and just in time). Nevertheless, it turned out that the intermediate representation used by LLVM (an assembly-level, machine-independent language) is an attractive target for other applications. Additionally, LLVM provides a number of C++ libraries for manipulating the intermediate representation, making it especially convenient to work with.
While the LLVM assembly is not the only such machine-independent, low-level language in existence, it appears to be the most suitable for the purpose at hand. While languages like the JVM and .NET assemblies are tailored for a specific programming language and its semantics (even though they have been adopted by other language runtimes later), the LLVM assembly is geared to more closely mimic the actual hardware architectures. Moreover, the existing support for assembly code manipulation through the LLVM-provided libraries is a significant advantage over the competition.
Interestingly, the formal semantics of LLVM IR have been partly worked out in [159], using Coq. The authors used Coq's code extraction to obtain a verified LLVM interpreter and compared it to `lli` as distributed with LLVM itself. Additionally, and more interestingly, they provide tool support for specifying IR-to-IR transformations directly in Coq (including correctness proofs based on the provided operational semantics), extracting those transformations and executing them as part of LLVM-based compilation.

2.2.2 Interpretation

When applying LTL model checking to software, we need to interpret the program under consideration as a Kripke structure. In fact, this is very simple and straightforward: the state of the system (an element of S) is described by the execution state of the machine running the program: the content of its registers and the memory accessible to the program. The transition relation then encodes the behaviour of the program: for all states a and instructions i in the program, $a \to b$ whenever the program counter captured in a points to i and b is the result of executing instruction i in b. For a sequential, deterministic program, the resulting structure is very boring: it is a linear sequence of snapshots of the program state, taken after executing each instruction. However, in a parallel program, multiple program counters may be active at any given time, and therefore some states a may have more than a single successor, making the structure much more interesting.
With this interpretation in place, the semantics of LTL properties are quite intuitive: the labels (atomic propositions) are properties of individual program states – they correspond to expressions over the current in-scope variables of the program. The formulae then, in effect, can make statements about temporal evolution of the values of variables of the program. If a global variable T in a controller for a temperature regulation system corresponded to, for example, the temperature measured on a sensor, it would be certainly sensible to demand $\mathcal{G}(T < T_{critical})$. On the other hand, if $in\_critical$ was a variable that is set to 1 whenever a critical section is entered, and reset to 0 when the critical section is left, we could demand that $\mathcal{GF}(in\_critical = 1) \wedge \mathcal{GF}(in\_critical = 0)$.
From how the interpretation is built, we can easily observe that the state space is constructed at an extremely fine-grained level, and while this allows us to get very precise results, it also severely compounds the state space explosion problem outlined above, placing significant stress on the implementation of the model checker.

2.2.3 Environment and Non-Determinism

In many cases, programs interact with their environment. Any of those interactions could lead to a number of different outcomes observable by the program, and we enumerate those as non-deterministic choice. This becomes a major problem quickly, as the choices compound exponentially and the reachable state space of the program inflates in a combinatorial explosion.
While part of this combinatorial explosion is unavoidable, in many practical programs, most of it will be due to noise, choices mostly irrelevant with regards to the programs' correctness. A typical example would be message payloads in a reliable network protocol: it is important that the message coming in is the same as the one coming out, but other than this simple fact, the actual data is unimportant. In some cases, the programmer will supply this information in some form: in a unit testsuite of the implementation, the payload will be mostly dummy data. There will be a few cases of "tricky" payloads

that could interfere with the protocol's data encoding. However, for the most part, the payloads would be 'foo's and 'bar's.
Basically, the programmer has intuitively abstracted away the payload in their testcases. Unfortunately, we can't rely on programmers doing this manual abstraction everywhere and for all kinds of data. While the programmer has unique insight into the structure of the program that allows him to spot abstraction opportunities, an automated tool has no such "intuition fairy" at its disposal. In terms of model checking, the programmer may provide a *model* for the environment, as part of the program that is given to the model checker. Despite this, it is desirable that the model checker can deal, at least in some cases, with open-ended programs, by accounting for any conceivable behaviour of the environment.

2.2.3.1 Arrays of Bytes

The tools that work with LLVM bitcode have a fairly limited understanding of what is going on in a program. Data pours in with no obvious structure: the program reads bytes from a file or a socket, usually in blocks through a fixed-size buffer. A faithful "non-deterministic" simulation of the program's environment would give a random-sized block of random bytes at each invocation of the `read` system call (bounded by the output buffer size, of course). The program will inevitably look at the data, mostly byte-by-byte and make decisions based on actual values of those bytes. Clearly, enumerating all those possibilities is extremely expensive, and most of that work is completely useless as well: vast majority of the inputs will quickly take the program down a path that rightly rejects the input as garbage.

2.2.3.2 Structured Input

While analysis of "raw programs" is desirable, it is also very ambitious. A middle ground can be struck by asking the programmer to provide verification "drivers" – basically, unit tests. However, in a traditional unit test, everything is fixed: all inputs are hard-coded in the test itself. An extension of this approach is to allow unit tests to specify ranges of values (say, an arbitrary integer between 0 and 10) instead of constants. This is something real-world programs often do, using various strategies to provide the inputs – whether by generating random inputs (like QuickCheck [44]), providing all "small" inputs (using some metric on the inputs; like SmallCheck [135]), using static analysis to come up with "interesting" cases to test, etc. Of course, it's also possible to use a model checker on such "open ended" test cases.
This compromise avoids most of the "garbage" inputs discussed in the previous section. Yet it's still easy to write unit tests that cause (exhaustive) model checkers to spin out of control, or to cause "deep" bugs that elude bounded model checkers, even at impractical unrolling depths. The favourite type of a bug with these symptoms is a failure at a boundary condition – overflowing an integral type, off-by-one errors near hard-coded application limits like static buffer sizes, message size limits, etc. When

debugging, programmers will often artificially drop these limits to a small number. However, this usually only happens after a problem is found accidentally – a tool that is trying to find the problem in the first place can't rely on the programmer to turn "deep" problems into "shallow" problems manually.

2.3 Program Semantics

In order to rigorously argue about programs, it is often useful to formalise their *meaning*, i.e. their *semantics*. Many different approaches have been invented to formally describe semantics of programs, and they usually work by assigning particular mathematical objects to programs. Both operational and denotational semantics follow this pattern, although they use a markedly different approach.

The difference between operational and denotational semantics mirrors the different view of programs as either state transformers (imperative programming) or as referentially transparent, possibly recursive expressions (functional programming, rewrite systems). As such, operational semantics focus on the state transformations involved in an imperative program, and they map program instructions to (total) functions that operate on program state. These functions are then most often constructed using inference rules describing how to transform a particular syntactic element in a programming language to a function of state.

In contrast, denotational semantics do not explicitly call for program state. In a purely functional (side-effect-free) language, a (denotational) semantic function (we will write $\mathcal{S}$ to mean the semantic function[4]) can assign functions to program expressions, effectively translating the program into operators of the meta-theory. This translation is compositional: a function assigned to a particular expression of the programming language is expressed in terms of the functions assigned to its sub-expressions.

While in operational semantics, the values involved in a program are captured explicitly as part of program state, in denotational semantics these values arise as *domains* (and *codomains*) of the functions assigned to expressions. In a statically typed programming language, the domains will most often consist of lifted types: say a value of type `Word32` will correspond to the domain (set) $W_{32} = \{\bot, 0, 1, ..., 2^{32}\}$. A function of type `Word32 → Word32` will correspond to some function $f : W_{32} \to W_{32}$. These functions are syntactically total, since non-definedness is expressed using an explicit $\bot$ symbol.

If we disregard type information, we can construct a single comprehensive domain for the entire programming language (including functions). This domain can then be ordered by "definedness": $(42, 15)$ is more defined than $(\bot, 15)$ or $(42, \bot)$, and these are in turn more defined than $(\bot, \bot)$ and $\bot$. Functions are ordered point-wise: $f \geq g \Leftrightarrow \forall x. f(x) \geq g(x)$. This ordering forms a CPO (complete partial order), with $\bot$ as the least element.

[4] The parameter of $\mathcal{S}$ is a program fragment and we will use Oxford brackets when applying $\mathcal{S}$: eg $[\![2 * 3]\!] = 2 \cdot 3 = 6$. Please note that 2, 3 and $*$ are parts of the programming language, whereas 2, 3 and $\cdot$ are part of the meta-theory. Moreover, if there is no risk of confusion as to which semantic function we mean, the Oxford brackets alone will be used, without the name of the function.

It is easy to see how to obtain semantics of simple expressions this way. However, a naive approach for deriving semantics of recursive definitions only using the inference rules has a fatal flaw: it fails to terminate, generating an infinite definition (even though this definition is otherwise correct). It is usually desirable that the semantic function is computable though[5], especially where we are interested in automated tools taking advantage of the semantics.
In order to obtain the requisite algorithm, we only need to realise that for a recursive definition f, there is a function g such that $[\![f]\!] = g^{\omega}$ (and function g can be obtained by using $\mathcal{S}$ on the right hand side of the recursive definition). Then, if g is monotonic[6], the function $g_i(x) = g \circ x$ is monotonic as well and $[\![f]\!] = g_i^{\omega}(\bot)$[7]. Since g_i is monotonic, f is obtained as its least fixed point and $\geq$ is a CPO, we know that $[\![f]\!]$ exists. [146]
This gives us a finite description of the semantics, and an algorithm to obtain such description.

2.3.1 Operational Semantics

As outlined above, operational semantics assign functions from state to state to program statements. There are three main challenges in formulating this kind of semantics: expressions, control flow and environment.
Operational semantics are normally presented, in fact, as a function of tuples (program, state) – a computational step is allowed to "rewrite" the program[8]. This efficiently deals with first two problems of the enterprise: expressions are simply rewritten "in situ", the same way term rewrite systems like lambda calculus work. The reduction rules are derived from semantics of the operators of the language. However, this approach is only applicable to very simple languages and is entirely inadequate for real-world programming languages. Even though it's not impossible to express jumps (`goto`) or recursion in this style, it is very awkward, and not very transparent. The rewrite rules become complex and error-prone, and it is hard to verify that the resulting semantics actually correspond to the intended semantics.
The way rewrite rules for the program portion of the tuple are formulated splits operational semantics into "small step" and "big step": small step semantics explicitly construct a sequence of states by only allowing elementary operations to happen in a single step. A program run corresponds to a long sequence of shallow derivations, each derivation mandating an atomic rewrite. In "big step" variant, there is only a single

[5] In the sense that the *description* of the semantics of the program can be computed. In other words, the semantic function in this context goes from the syntax of the programming language to the syntax of the meta-theory. The meta-theory itself is usually not decidable.
[6] Clearly, not all computable functions are monotonic. However, functions arising as semantics of the non-recursive fragment of many languages are naturally monotonic.
[7] $\forall x.\bot(x) = \bot$
[8] In effect, the program text becomes part of the state and the machine "keeps track" of execution of the program by rewriting it. The program text has in a way become a proxy for the program counter.

derivation tree, and state transitions are implicitly encoded in the derivation tree. Neither of those styles is suitable for general-purpose imperative programming languages, although the "small step" style is more useful than the "big step" style. Hence, let's consider an evolution of small-step semantics more useful in the real world, modelled not after an abstract rewriting machine but an abstract von-Neumann-style machine.
In a non-rewriting abstract machine, the "normal" operational approach to expressions no longer works, for two reasons: the program is now immutable, and all the state of the program, including intermediate values arising from sub-expressions, need to become part of the explicit state. Probably the best approach to this is to use three-address code representation of the expressions. This transformation is very straightforward and solves both problems: the abstract machine state is reduced to a single datum – which statement is to be executed next. This corresponds to a program counter as it exists in many abstract and most real machines. The problem of intermediate values is solved by assigning a fresh name to each intermediate result, making it explicit.[9]
In the rewriting-based system, the simple control flow that can be reasonably captured – `if` statements and `while` loops – are realised straightforwardly. The advantage of that is that the semantics capture these constructs at a high level. Especially semantics of while loops are expressed clearly – if the loop condition is true, the body is executed, followed by a copy of the original while loop. Unfortunately, this only works in strictly structured programming languages. Using program counters instead is far less intuitive in some sense, but allows capturing arbitrary control flow uniformly: conditional branching (`if` statements), local jumps (`goto`), recursion and even exception handling and non-local jumps (akin to `setjmp/longjmp` or asynchronous signals). It has the added advantage of mimicking a compiler more closely (even most modern real-world interpreters shun term rewriting).[10]

2.3.2 Environment vs Operational Semantics

The last challenge remaining in formulating operational semantics is the environment. Both input/output and scheduling in a multi-threaded program are a source of non-determinism in program behaviour, which is however hard to capture in semantics as we have defined them so far. Computation trees provide one possible answer: each "small operational step" gives a set of possible result states, instead of just one.[11] This

[9] This translation can be done at the level of the source text, or it can be made part of the semantic function. The latter is in some way more "pure", although it makes the semantic function more complicated. Nevertheless, since LLVM is in fact a form of three-address code, we won't elaborate this topic any further.

[10] In some sense, implementation of a language – be it a compiler or an interpreter – is a description of the language's semantics. While it is often bogged down in uninteresting implementation details, it is also an exceedingly formal description. After all, it can be executed by a computer. We are however interested in pen-and-paper semantics as a simplification, and as a basis for arguing about programs. A traditional compiler is too complex for this role. However, see also CompCert [114], a verified compiler for a structured subset of C. The semantics of the source language and of all the intermediate languages are specified in Coq.

[11] This is, in fact, how explicit-state model checkers represent asynchronous systems.

has a major downside though: a loss of composability – it is no longer possible to derive semantics of sub-programs and compose them to obtain the semantics of the entire program. This is a major flaw, and a reason to avoid this style.
Instead, we extend the function that represents program semantics with a second parameter, representing the environment. The function of program state and the environment then works somewhat like an oracle machine: the environment can be queried for values. Unlike with an oracle though, obtaining a value from the environment also changes the environment. We could imagine a function like $query : E \to V \times E$, where V is a suitable domain of program values and E is the set of possible environment "states". Then, for some program p, we have $[\![p]\!] : \Sigma \times E \to \Sigma \times E$. The definition of $[\![p]\!]$ makes use of *query* to obtain data from the environment; it has no direct access to values from E, as these are opaque objects. The existence of *query* is pre-supposed, but its definition is not given, since the representation of the environment is not important from the point of view of program semantics. The existence of E (the set of all possible environments) is assumed as well.
The use of *query* in the derivation rules for building $\mathcal{S}$ would look somewhat like this (p refers to program text, pc refers to the program counter variable):

$$\frac{\mathrm{p}_{\sigma(\mathrm{pc})} \equiv \mathrm{x} := \mathrm{read\_stdin} \wedge query(\varepsilon) = (\mathrm{v}, \varepsilon')}{\langle \sigma, \varepsilon \rangle \longrightarrow \langle \sigma[\mathrm{pc} \mapsto \sigma(\mathrm{pc}) + 1, \mathrm{x} \mapsto \mathrm{v}], \varepsilon' \rangle}$$

Besides the explicit manipulation with program counter, this rule doesn't look so bad. However, in multi-threaded program semantics, scheduling would also become explicit. In this case, σ would need to encode a list of active threads and a flag indicating whether an execution or a scheduling step is active, as well as the thread selected for running. A scheduling rule would then look roughly like this

$$\frac{\sigma(\mathrm{scheduling}) \wedge (\mathrm{t}, \varepsilon') = query(\varepsilon) \wedge \mathrm{t} \leq \sigma(\#\mathrm{threads})}{\langle \sigma, \varepsilon \rangle \longrightarrow \langle \sigma[\mathrm{scheduling} \mapsto \mathrm{false}, \mathrm{thread} \mapsto \mathrm{t}], \varepsilon' \rangle}$$

Clearly, all computation rules would need to add $\neg\sigma(\mathrm{scheduling})$ to their premises and set scheduling to true in the resulting state. They would also need to refer to $\sigma(\mathrm{pc}_{\sigma(\mathrm{thread})})$ as the program counter.
Plainly, a system of rules as complex as this, fully spelled out, is rather impractical for human use, especially for any realistic programming languages. Nevertheless, an extra level of abstraction can be added fairly easily – a small set of syntactic shortcuts can be used to write a relatively succinct set of derivation rules, hiding the bookkeeping related to program counters and thread scheduling. Still, formulation of semantics such as this is mainly useful in implementing the system – be it a compiler or an interpreter – and as a part of a model checker.

2.3.3 From Operational to Denotational Semantics

Now that we have formulated operational semantics that act on states (and on environment), we would like to return to a higher-level view. As outlined earlier, denotational

semantics assign mathematical objects to programs, in a compositional manner. We now posses operational semantics that allow us to turn a program statement into a state transformer and a program into a series of such transformers. Sequences of transformers give rise to sequences of intermediate states, forming execution traces. These traces are the main subject of our analysis. The semantic function for a program, then, gives us a set of all possible execution traces – we know that our operational semantics make it possible to enumerate this set of traces, as long as the program always terminates.
The high-level structure of this semantics is quite important, though, regardless of how it is obtained. There are many ways to compute and to represent this set of traces, ranging from a simple graph of possible states (merging identical states both from a single trace and from multiple traces into a single node), through various symbolic and approximate representations. We will deal with those in **Section 2.4**.
Hence, in the spirit (if not the letter) of denotational semantics, the mathematical object we will assign to programs is a function from an environment (as defined in **Section 2.3.2**) to a sequence of states (a trace). In this model, the environments are a generalisation of inputs. By lifting all non-determinism into a parameter, we obtain a functional description of the program, as opposed to procedural (imperative). This is intimately connected to how model checkers work in practice.

2.4 Symbolic Methods

We have outlined the basic idea common to all symbolic methods, namely use of a symbolic representation for a set of system states, in **Section 2.1.1**. In this section, we will further explore various representations and the model checking methods based on these representations.

2.4.1 Binary Decision Diagrams

The first data structure in widespread use for representing sets in symbolic model checking is known as a Binary Decision Diagram (BDD), a compact tree structure describing a set of bit vectors. Early symbolic model checkers tailored towards verification of hardware designs applied this simple but efficient structure with extreme success [38]. The way this has been exploited is that in the model checker, the set of reachable states is represented as a single BDD, and the transition function of the system is encoded as a BDD as well. Finally, encoding the set of initial states and iteratively unioning it with its own image under the transition function yields the entire reachable state space. A safety criterion can then easily be checked by evaluating the property over all reachable states, possibly in every iteration to obtain an on-the-fly algorithm.
For temporal properties, more elaborate algorithms are required. The logic usually associated with BDD-based symbolic model checking is CTL, as those model checkers have been usually targeted at hardware verification. Additionally, hardware models are usually synchronous, unlike software. In this domain, BDDs are a particularly good fit, as they represent logical circuits very well. The main weakness of BDDs, namely integer

multiplication and arithmetic derived from multiplication, is less relevant in hardware, as multiplication circuits are complex and as such expensive in terms of silicon – in contrast to software, where multiplication is ubiquitous.
The main algorithm in use for symbolic CTL model checking is OWCTY [73] – it is a good fit, since it is readily expressed in terms of set operations. To our best knowledge, the only symbolic LTL model checking tools that represent the reachable set as a BDD use the "liveness as safety" approach [26]: compared to automata-based LTL model checking, this incurs an additional exponential penalty in the size of the state space. Since state space size and tractability are only loosely related in symbolic model checking, whether the extra complexity or whether the compact symbolic representation dominates is highly model-dependent.

2.4.2 Abstraction & Refinement

In the software world, BDDs did not yield nearly as much success as with hardware. Implementation of even basic integer arithmetic in terms of BDDs is very complex and not very space efficient. While integer addition is readily and efficiently expressible in terms of BDD operations, this is not the case for multiplication: in fact, in terms of BDDs, multiplication is an inherently exponential operation, regardless of variable ordering [37]. A similar problem exists with other canonical forms, eg. for functions over bit vectors. Since multiplication is essential in almost all software, it is hard to imagine relegating it to a second-class status as compared to addition.
Instead, abstractions, and most prominently predicate-based abstractions, have been widely used in software verification. Even though abstraction-based model checking is generally regarded as quite successful, it has failed to become a mainstream technology in software development. Probably the closest we have seen is Microsoft's driver verification toolkit [4]. However, driver development is a niche market in the wider context of software industry, even if we only consider software written in C and C++. In **Chapter 7**, we will explore some of the options that might enable wide-scale abstraction-based verification of programs; a key ingredient in this effort is taking advantage of pre-existing technology which works with concrete programs.

2.4.3 Abstract Interpretation

Abstract interpretation is a very general term: it is not particularly constrained to interpretation in the sense of a program interpreter; instead, it refers to interpretation as providing semantics to the program. In other words, abstract interpretation does not change the syntax of a program being interpreted abstractly, but it changes what the program means, in a way that allows us to infer important insight about the *original* or *concrete* behaviour of the program.
Denotational semantics of a programming language are expressed using a *semantic function* (see also **Section 2.3**), mapping program syntax to appropriate mathematical objects: usually partial functions. Such partial functions over a partially ordered set

form themselves a complete partial order (cpo) according to their point-wise "definedness". In case of imperative programs (i.e. the case most interesting for model checking), these partial functions map from program states to program states.
Abstract interpretation then essentially means replacing the *semantic function* of a particular programming formalism with an abstracted version, and arguing about program behaviour in terms of the new (abstracted), denotational semantics. In order for such arguments to be sound, the abstraction must over-approximate the concrete behaviour of the program. There is a formal relationship between the *concrete* and the *abstract* domains. Let $\mathcal{C}$ be the concrete semantic function, and $\mathcal{A}$ the abstract one. Then, $\mathcal{C}[\![P]\!]$ is the concrete semantics of P, a partial function from some *concrete domain* C to itself. Likewise, $\mathcal{A}[\![P]\!]$ is the abstract semantics of P. Let $f = \mathcal{C}[\![P]\!]$ and $f' = \mathcal{A}[\![P]\!]$. In order to talk about validity of an abstraction – i.e. its ability to give answers that are relevant in the concrete domain – we need to establish a connection between f and f'. This connection is usually described in terms of two functions between the abstract and the concrete domain, labelled α and γ, for abstraction and concretisation, respectively. Together, they describe how to translate concrete values into abstract values and back again. While at a first glance, we would like $\gamma \circ \alpha$ to be an identity, this would defeat the purpose of our exercise. In fact, even demanding (α, γ) to be a Galois connection[12] is often too stringent, although it can be satisfied in specific instances (namely those where best abstractions exist). To guarantee that a particular abstraction is over-approximating, it is enough that $(f \circ \gamma)(x) \leq (\alpha \circ f')(x)$ (where x is taken from $\mathrm{dom}(f')$).
While some analysis tools are concerned (in addition to these simple criteria) about the existence and computability of fixed points in the abstract domain (in order to summarise loops or recursion) we are not directly interested in those issues. A future extension of this work may entail more elaborate analysis of recursive program constructs in order to improve the efficiency and/or precision of the abstraction. With this in mind, we will generally try to use abstractions that have the requisite properties – moreover, most research to date deals with such abstractions.

2.4.4 Symbolic Execution

In the domain of testing, there is a related trend for symbolic evaluation in place of concrete (or explicit) execution. The problem formulation in testing is somewhat different from that of model checking – in the case of symbolic execution, the goal is to construct a set of inputs for testing that cause all branches to be explored [52]. To this end, a program is evaluated with input symbolic values and whenever branching is encountered, the set is split using the branching conditional into two, each causing one of the branches to be taken (in some cases, either of those sets can turn out to be empty).

[12] I.e. requiring that $\forall a, b.\alpha(a) \leq b \Leftrightarrow a \leq \gamma(b)$, or, in other words, that α and γ are adjoint.

The input value sets are usually represented each by a set of symbolic constraints on input variables, effectively implementing a specific kind of abstract interpretation. In the context of LLVM, KLEE [39] is an example of a tool based on symbolic execution, automatically generating test cases ensuring high coverage.
Clearly, since symbolic execution considers each execution path separately, it has to deal with exponentially many paths through the program, resulting in a so-called *path explosion*. Various methods for dealing with this problem have been proposed, including composability and use of SMT solvers [1], parallelisation [142] and directed search [116].

2.4.5 Bounded Model Checking

A major renaissance of symbolic methods in model checking coincides with advances in satisfiability checking: SAT and later SMT solvers had become sufficiently powerful to make a formula-based symbolic representation feasible. However, bounded model checking constitutes a marked departure from the earlier exhaustive approaches: a straightforward bounded model checker cannot give a positive answer unless all the loops in a program are bounded by a fairly small constant.
Clearly, bounded model checking can be augmented to make it possible to get a positive answer for correct programs that execute endless (or very long) loops. Many heuristics have been invented and implemented to eg. infer loop invariants – and with the right invariant, the loop can be replaced by a finite formula in construction of the SAT/SMT representation, instead of its (possibly infinite) unrolling.

2.4.6 Set-based Model Checking

Yet another symbolic approach to model checking of programs is closely related to explicit-state model checking but builds on the ideas from early symbolic model checkers. Namely, the gist of the technique is to follow the same exact, exhaustive procedure as an explicit-state model checker would, but when possible, take advantage of regularity in data. This is especially useful for open programs – those that read arbitrary input data, from a possibly large domain. In those cases, it is easy to establish a clear relationship between a large number of states: namely those that only differ in the value of variables that somehow depend on the input. The idea is to find a compact encoding for sets of values which also admits efficient computation of their images under operations encountered in programs. Equipped with such a set representation, a large number of related states can be replaced by a single state, where some variable values are represented as sets [5], and operations on these sets are carried out in bulk. Additionally, when a value of particular variable affects program control flow, it must be possible to prune the set based on that particular control flow decision – if a program enters a *then* block of a conditional statement, guarded by, say $x < 5$, the set representation of x at that point must be intersected with set of all $x < 5$. For this reason, it is additionally required that such intersections can be efficiently computed. Finally, it

must be possible to efficiently compare such sets: depending on a particular model checking algorithm, either equality or subsumption may be required.
The large number and often conflicting nature of those requirements makes it particularly hard to find suitable set representations. However, as long as the bulk (set image) operations are significantly faster than computing the image "one by one" (by applying the primitive operation to each set element and accumulating the results) and as long as the sets represented this way are significantly more compact than a list of elements would be, this approach can save considerable resources. For many common domains, fully explicit representation of all their possible values is prohibitively expensive: application of this set-based approach can boost the chances of successful verification whenever arbitrary values from large domains enter the computation.

2.5 Explicit Methods & Scalability

Explicit-state model checking techniques, while very appealing due to their full automation and simplicity, have serious scalability limits if implemented naively. Even though computer hardware is advancing rapidly, automatic model checking can easily exceed the capacity of a single computer, or even a sizeable cluster. Many approaches have been invented to fight this scalability problem, although they are often hard to combine and such combinations are subject of ongoing research. Moreover, the scalability issue has two related, but still separate aspects: space and time requirements.
There are two basic approaches to improve *memory* scalability of model checking to larger systems: making the model smaller (which comprises state space *reductions* [127 and 157] and *abstractions* [49]) and making the capacity of the model checker bigger (which comprises distributed memory and clusters [41, 43 and 75], leveraging external memory like rotational hard drives or solid-state technology [19], state space compression [76 and 86], or combinations thereof [20]). A more complete survey of the state of the art in memory scalability of explicit-state model checking may be found in [126].
While the *runtime* aspect can benefit from the memory-oriented optimisations listed above, the effect may not be as pronounced, or may even be reversed by increase in computational complexity and/or overhead. We have conducted intensive research of using shared-memory *parallelism* for improving runtime performance of the model checker DIVINE, both in terms of improved algorithms [15] and improved engineering [9]. Additionally, [111] comprises a substantial, recent collection of material on scalability of explicit-state model checking on parallel shared-memory machines.
In addition to the above techniques, a number of *heuristic* approaches have been proposed, that cannot give a definitive answer to the verification problem, but, using significantly reduced computational resources, can give a "very probably correct" answer. For verification purposes, hash compaction [143 and 156] is the most suitable such heuristic, with the additional advantage of allowing us to quantify the margin of error. Other, more space- or time-efficient techniques exist which are more suitable for falsification efforts ("bug hunting"), including bitstate hashing [85], heuristic searches [63] or random walks [141].

2.5.1 Parallelism and Distributed Memory

Both these approaches serve to increase the raw power available to the model checker, in terms of computing power and available fast random-access memory, and that way, broaden the scope of models that can be successfully verified. However, the approach is not without its own problems and challenges. The best sequential algorithms for implementing an automata-based LTL model checker (primarily Nested DFS) cannot be fully parallelised, being based on depth-first postorder [132]. This led to the development of a number of different algorithms, with different trade-offs, that can be used in distributed-memory systems – the main rationale was that access to more powerful hardware will outweigh the sequential disadvantage of these algorithms. The current state of the art parallel (and distributed) algorithm for automata-based LTL model checking is an improved variant of explicit-state version of One-Way Catch Them Young (OWCTY) [15], which is in turn based on earlier parallel algorithms [33 and 42]. More recently, speculative parallel execution of Nested DFS (Nested DFS) on shared-memory parallel hardware has seen substantial success in the model checking community, in the form of MC-NDFS [69], CNDFS [68] and related algorithms. Finally, a special case of Nested DFS for a 2-core computer can achieve non-speculative parallelisation [89]. However, none of the parallel Nested DFS modifications can be directly used for distributed LTL model checking.

2.5.2 Compression

An important technique that can contribute to memory efficiency of explicit-state memory is (lossless) state space compression. The oldest and simplest method was to use a generic data compression algorithm (Huffman coding, arithmetic coding, etc.) to compress individual state vectors before storing them into memory [76 and 90]. These approaches only minimally exploit the redundancy *between* different states, which is usually much higher than the redundancy *within* a single state vector.
In this respect, a better method has been proposed in [86], where the state space vector is decomposed and each slice of the vector is hashed separately. This exploits the fact that many state vectors contain parts that are identical and also much longer than a single pointer – hence, storing a pointer to a separately hashed slice is more memory-efficient than storing the duplicated area repeatedly. While this idea is in a way a specialisation of otherwise very generic and well-known dictionary-based compression (as employed by the commonly used LZ77 [160] algorithm), it has some special properties that make it more interesting for model checking: namely, the construction of the "dictionary" makes it easy and efficient to hash the compressed states and compare them for equality – neither of those steps needs to decompress states already stored.
The one-level scheme proposed in [86] has been improved upon by [28], making it fully recursive. It also removes the requirement that the compression algorithm knows specifics about the state vector layout. This recursive approach has been further

adapted for parallel model checking in [110]. One downside of this implementation is a requirement for a fixed-size, pre-allocated hash table.
We use a similar scheme, but we re-introduce *optional* state vector layout awareness into the compressor, we use generic n-ary trees instead of binary and we use resizing hash tables in the implementation. More details about our approach can be found in **Section 3.6**.

2.6 Algorithms for Accepting Cycle Detection

An efficient parallel solution of many problems often requires approaches radically different from those used to solve the same problems sequentially. Among classical examples are list rankings, connected components, and depth-first search in planar graphs.
In the area of LTL model checking, the best known enumerative *sequential* algorithms based on accepting cycle detection are the *Nested DFS* algorithm [55 and 92] (implemented, e.g., in the model checker SPIN [87]) and *SCC-based algorithms* originating in Tarjan's algorithm for the decomposition of a graph into strongly connected components (SCCs) [145]. However, both algorithm types rely on the inherently sequential depth-first search post-order. This property of the algorithms makes them difficult to adapt to parallel architectures. The SPIN dual-core algorithm [89] is a special case, where the nested (second) search is independent of the outer (first) search. Nevertheless, each of the searches is, in itself, executed serially – therefore, the algorithm cannot be generalised to more than 2 cores. For distributed-memory model checking, different techniques and algorithms are needed. While in shared memory, the CNDFS algorithm is quite successful, distributed-memory algorithms executing in shared memory are competitive, offering a different set of trade-offs.
First, let us define a concise way to describe the input and output of an accepting cycle detection algorithm:

def. 2.1 The accepting cycle problem instance M is a tuple (V, E, A, I) where V is a set of vertices (states), $E \subseteq V \times V$ is a set of edges (transitions), $A \subseteq V$ is a set of accepting states and $I \subseteq V$ is a set of initial states. ■

Further in this section, we will discuss reachability analysis, then proceed to the "staple" sequential algorithm, Nested DFS, and its multi-core adaptations. Finally, we will look at some of the most successful distributed-memory parallel LTL algorithms available: MAP and OWCTY.

2.6.1 Reachability Analysis

Unlike LTL model checking, reachability analysis is a verification problem for which an efficient parallel solution is available. The reason is that the exploration of the state space is independent of the search order. This makes the algorithm easy to implement

on parallel architectures with relatively good efficiency out of the box (assuming that efficient parallel primitives for given architecture are correctly employed – we will discuss these in more detail in **Chapter 3**).

---

alg. 2.1 Reachability can be expressed as a simple stateful algorithm over sets. Its state can be captured by two sets, open and closed:

```
data C = C { _open :: [S], _closed :: [S] }
```

The main procedure is a simple fixed-point loop. While it could be implemented by keeping two copies of the "closed" set and compare them for equality after each iteration, this is less efficient than keeping an explicit "open" set (even more-so in practical implementations, where the open set is usually represented as a queue). When the open set becomes empty, the algorithm escapes the loop by invoking the "done" continuation.

```
reachability' :: M → Computation [S] C
reachability' m@(vs, es, _, is) done =
  forever $ do
    c ← get closed
    o ← get open
    when (empty o) $ done c
    closed ⇐ (o ∪ c)
    open ⇐ nub [ s | (v, s) ← es, v ∈ o, s ∉ c ]
```

At the outset, we set the open set to be the set of initial states and the closed set starts out empty:

```
reachability :: M → [S]
reachability m@(_, _, _, is) = compute (reachability' m) $ C is []
```

2.6.2 Nested DFS

Nested DFS is the staple sequential algorithm for accepting cycle detection, and as such, for explicit-state LTL model checking (cf. **Section 2.1.2**). The algorithm proceeds by first exploring the graph from its initial vertices in a depth-first order. Upon backtracking through an accepting state, a nested search is started: this nested search uses its own closed set, which is shared by all instances of the nested search, but not with the main (outer) search. This closed set is usually implemented as a single-bit flag in the representation of a vertex (the same technique is used for the outer search, giving theoretical overhead of 2 bits per vertex, although achieving this overhead in practice is not trivial due to memory layout issues).

---

alg. 2.2 Most graph algorithms are stateful, and Nested DFS is not an exception. The state of the algorithm is captured by the following data type:

```
data C = C { _c_inner :: [S], _c_outer :: [S], _seed :: Maybe S }
```

We keep two "visited" sets, one inner and one outer, and possibly a seed value (it is initialised at the outset of the nested search). The main computation of the search just runs the outer phase from each initial vertex in turn. If no sub-search invokes the "done" continuation, the search has completed without discovering an accepting cycle and returns `Nothing`:

```
ndfs' :: M → Computation (Maybe S) C
ndfs' m@(vs, es, as, is) done = do for is outer; return Nothing
```

both phases of the search have the same structure, the only difference being that the outer search does its check in the "post" position (since the search is depth-first, this means in the DFS post-order), while the inner search runs its check ("has an accepting cycle closed?") in pre-order.

```
  where
    visit c pre post (v :: S) = callCC $ \skip → do
      pre v
      seen ← get c
      when (v ∈ seen) $ skip ()
      c ⇐ (seen ∪ [v])
      sequence [ visit c pre post s | (v', s) :: E ← es, v ≡ v' ]
      post v
```

Whenever the outer search encounters an accepting state, it sets the seed and runs the inner search.

```
    outer = visit c_outer noop $ \v →
      when (v ∈ as) $ do
        seed ⇐ Just v
        visit c_inner inner noop v
```

Finally, the inner search short-circuits the computation by invoking the "done" continuation if it finds an accepting cycle. This makes the algorithm "on the fly", arranging for an early termination when a counterexample is found. Please note that in actual implementations, the inner search may be made more efficient by checking whether it has encountered a state that is on the search stack of the main (outer) search.

```
  inner v = do
    visited ← get c_inner
    Just s  ← get seed
    when (s ∈ visited ∧ v ≡ s) $ done (Just v)
```

Finally, the entire Nested DFS procedure simply consists of running the computation with empty "visited" sets and an empty seed.

```
ndfs :: M → Maybe S
ndfs m = compute (ndfs' m) $ C [] [] Nothing
```

2.6.3 Dual-Core Nested DFS

The paper [89] presents an extension of Nested DFS to two cores, on the observation that once a nested search is entered, it cannot backtrack into the main search, and hence can be safely executed in parallel. This algorithm is available in DIVINE, but only used if explicitly requested, using `divine verify --nested-dfs -w 2`.
The algorithm is a straightforward modification of the sequential Nested DFS, where the call of the inner search is implemented in terms of a promise that is processed in parallel while the main search continues. There is only one thread that delivers on promises created in this way.

2.6.4 CNDFS

The CNDFS algorithm represents a convergent evolution of two earlier multi-threaded algorithms based on nested depth-first search, LNdfs [108] and ENdfs [69]. While the original algorithms are fairly complicated, the combined algorithm is surprisingly simpler than either of its predecessors. A yet earlier idea is to run a number of non-synchronised, randomised instances of Nested DFS on the same graph. In the case where there is no counterexample (accepting cycle) in the graph, this scheme does not improve anything – it does however lead to improvements in cases where accepting cycles are present in the graph. Since depth-first searches are subject to "luck" – they may descend into a deep part of the graph where there are no counterexamples if they are "unlucky" or conversely to a shallow area with a counterexample (if they are "lucky"). The randomisation in this unsynchronised, *swarm* approach [91] helps to push the odds of at least one search becoming lucky higher. Once any of the searches hits a counterexample, the entire swarm can be stopped.
The approach of both LNdfs and ENdfs is similar to the *swarm* approach, but both these algorithms try to also share some work in cases where there is no counterexample in the state space. Where the LNdfs approach concentrates on sharing results of the nested search – avoiding multiple nested visits of a particular area of the graph, possibly pruning the outer search in late-coming threads as well; conversely the EMdfs approach tries

to share the results of the outer visit by speculatively sharing the results of computing the post-order and detecting cases where it was violated by the reordering, which are then repaired. While a naive combination of both algorithms, dubbed NMCNdfs [109], combines their strengths (where work is shared successfully by multiple threads) it also combines their weaknesses, resulting in a very complex and a relatively memory-intensive algorithm (requiring nearly a full byte of flags per state per thread).
A better algorithm[13] can be obtained by replacing the repair procedure from EMdfs by a wait. Whenever a nested search is launched, since it may be running out of post-order, it could encounter a state that was not yet visited in the outer search. In those cases, instead of proceeding and trying to repair the post-order later, the algorithm simply waits for those states to be marked as visited in a nested search by another thread – clearly, there must be a thread to do so, since the current seed is out of post-order and this must have been caused by another thread. That thread will therefore also run the nested search on those (earlier in the post-order) states.
The final algorithm, CNDFS, is presented in more detail in [68], along with detailed correctness and termination proofs.

2.6.5 OWCTY

The next algorithm is called One-Way Catch Them Young – or OWCTY for brevity. It has been introduced for explicit-state model checking in [42], and is the main multi-core LTL algorithm in DIVINE (with later extensions, as discussed in **Section 2.6.8**). The algorithm is executed in passes, each of them consisting of a number of steps.
The scheme of the main loop basically describes how to convert any simple cycle detection algorithm into an algorithm for accepting cycle detection. The usual simple cycle detection algorithm employed here is based on topological sort. The elimination step uses this algorithm to remove all nodes that do not lie on cycles. Of course, the cycle detection algorithm does not discriminate accepting and non-accepting cycles, which is why we need to also exclude states that are not reachable from an accepting state: these clearly cannot lie on an accepting cycle.

---

alg. 2.3 The (mutable) state of the algorithm can be kept down to a single set of states.

```
data C = C { _s :: [S] }
```

The main algorithm loop executes the passes until a fixed point is reached. The result is a set of states that, if not empty, contains an accepting cycle.

[13] At least as far as model checking is concerned, since we only have experimental evaluation to back up the relative performance of various parallel algorithms for accepting cycle detection.

```
owcty' :: M → Computation [S] C
owcty' m@(vs, es, as, _) done = forever $ do
  s₁ ← get s
  let s₂ = reachability (vs, es, as, s₁)
  when (empty $ s₂ ∩ as) $ done []
  s₃ ← callCC $ elimination s₂ es
  when (s₂ ≡ s₃) $ done s₂
  s  ⇐ (s₃ ∩ as)
```

The elimination algorithm (implemented using topological sort, as mentioned above) is as follows:

```
elimination :: ∀s. [S] → [E] → Computation [S] s
elimination vs es done = do
    when (empty tail) $ done vs
    elimination [ v | v ← vs, v ∉ tail ] es done
  where tail = [ t | t ← vs, empty [ () | x ← vs, (x, t) ∈ es ] ]
```

At the outset, we set the open set to be the set of initial states and the closed set starts out empty:

```
owcty :: M → [S]
owcty m@(_, _, _, is) = compute (owcty' m) $ C is
```

It can be seen that there is a reasonable amount of available parallel work: each state in *tail* can be processed independently of all the others in each iteration. Unfortunately, there is only limited parallelism available across the iteration boundary – we have to wait till all predecessors of a state are processed before we can process the given state. Looking at serial complexity of the algorithm, we should discuss two cases: a weak graph, and an arbitrary graph. For the weak case, we ought to use a different main loop. In a weak graph, there are no cycles that would contain both accepting and non-accepting states – therefore, non-accepting states can be automatically discarded from cycle detection: only a single pass of reachability needs to be done through non-accepting components (to detect the possible neighbouring accepting components). For the accepting components, we first do a single reachability pass to discover all the vertices belonging to the given component and when this is done, execute a single elimination pass on that component. The component contains an accepting cycle iff the elimination pass returns a set of vertices that is a proper subset of the component's vertex set.

---

alg. 2.4 Unlike most algorithms for accepting cycle detection that we have presented here, OWCTY for weak graphs is a linear algorithm that operates in a single pass. Since we

re-use the elimination algorithm from **Algorithm 2.3**, the formulation of weak OWCTY is extremely simple:

```
owctyweak :: M → [S]
owctyweak m@(_, es, as, _) = compute (elimination as' es) ()
   where as' = as ∩ reachability m
```

All that we do is compute the set of all reachable accepting states and execute elimination on this set – since all accepting states come from neverclaim SCCs that only contain accepting states, no accepting cycle can be interrupted by a non-accepting state. Finally, `elimination` returns a non-empty set whenever a cycle exists consisting entirely of vertices given by `as'`, the set of all reachable accepting states in the graph.

2.6.6 MAP

The name of this algorithm is an acronym for **Maximal Accepting Predecessors**. It has been initially designed for distributed memory systems, in [33 and 35]. The algorithm is based on the fact that every accepting vertex lying on a cycle is its own predecessor (and this cycle, containing an accepting vertex, is an accepting cycle). An algorithm that is directly derived from this idea would require expensive computation as well as space to store all proper accepting predecessors of all (accepting) vertices. An improvement over that, the MAP algorithm stores only a single representative of all proper accepting predecessor for every vertex, chosen to be *maximal* accordingly to a presupposed linear ordering $\prec$ of vertices (given, for example, by their memory representation). Clearly, if an accepting vertex is its own maximal accepting predecessor, it lies on an accepting cycle. On the other hand, it can, unfortunately, happen, that all the maximal accepting predecessors lie outside accepting cycles. In that case, the algorithm removes all accepting vertices that were the maximal accepting predecessors of any vertices in the previous pass and recomputes the maximal accepting predecessors. This is repeated until an accepting cycle is found or there are no more accepting vertices in the graph.

---

alg. 2.5 The MAP algorithm maintains a set of vertices, `shrink`, which contains maximal accepting predecessors known to lie outside of any cycle. These are disregarded in later passes. It also keeps a structure similar to an "open set", although instead of single vertices it contains tuples, each vertex accompanied by its best known maximal accepting predecessor. Finally, a function from vertices to vertices called `mapfun` maintains the current mapping from a vertex to its maximal accepting predecessor, for all vertices (this function is progressively updated using the more recent data present in the `open` data structure).

```
data C = C { _shrink :: [S], _open :: [(S, S)], _mapfun :: S → S }
```

As stated earlier, MAP is a multi-pass algorithm. This is because a single pass may, as outlined above, fail to find an accepting cycle: all maximal accepting predecessors may lie outside of accepting cycles. To this end, the `shrink` set is maintained throughout the computation. After a given pass, the set contains all maximal accepting predecessors that were found in that pass. If no accepting cycles were found, this means that all these vertices lie outside of cycles and the main algorithm can remove these from the accepting set and start a new pass.
Each pass then consists of $\mathcal{O}(|E| \cdot |A|)$ steps, each of which updates a single BFS row of states with new information (most importantly the maximal accepting predecessor function, called `mapfun` in the source code). A single step is implemented as follows:

```
step :: M → C → ([S], C)
step m@(vs, es, as, is) (C shrink₁ open₁ map₁) =
  (map fst cycle, C shrink₂ open₂ map₂)
 where
  shrink₂ = [ x | x ← shrink₁ ∪ map fst open₁, map₁ x < x ]
  upd = [ (x, if x ∈ as ∧ x > map₁ x then x else map₁ x)
            | (x, p) ← open₁ ]
  map₂ x = if (x, x) ∈ upd then x else map₁ x
  next f = [ (w, p) | (v, w) ← es, (x, p) ← upd, x ≡ v, f p w ]
  cycle = next $ \p w → p ≡ map₂ w
  open₂ = next $ \p w → p > map₂ w
```

Each pass then consists of updating the state until either a fixpoint is reached (no new map values can be propagated) or an accepting cycle is found.

```
pass :: M → Computation [S] C
pass m@(vs, es, as, is) done = do
  open ⇐ zip is is
  forever $ do
    state₁ ← gets id
    let (cycle, state₂) = step m state₁
    put state₂
    when (cycle ≠ [] ∨ _open state₂ ≡ []) $ done cycle
```

Finally, the algorithm is executed until either `shrink` encompasses all of A, in which case no accepting cycles exist in the graph, or when a pass finds an accepting cycle.

```
mapalg :: M → [S]
mapalg m@(vs, es, as, is) = compute loop $ C [] [] (const 0)
 where loop done = forever $ do
         shrink₁ ← get shrink
         if empty shrink₁ then done []
                          else pass (vs, es, as \\ shrink₁, is)
```

The overall time complexity of the algorithm is in $\mathcal{O}(a^2 \cdot m)$, where a is the number of accepting vertices and m is the number of edges. The m factor comes from the relaxation along edges, while one of the a factors comes from the inner pass and the second a comes from the number of outer iterations of the algorithm.
One of the key aspects influencing the overall performance of the algorithm is the underlying ordering of vertices used by the algorithm. Computing the optimal ordering is however difficult to parallelise, hence heuristics for computing a suitable vertex ordering are used.

2.6.7 On The Fly Execution

Parallel techniques for both symbolic and explicit state approaches have been considered. While the symbolic set representations, which often employ canonical normal forms for propositional logic (BDDs, for example), have been a breakthrough in the 1990s (with the capacity to handle spaces of the size 10^{20} and beyond), they often turned out to not scale well with problem sizes. Moreover, the success of their application to a given verification problem cannot be estimated in advance, since no known metrics for the system size have proved to be useful for such estimates. Finally, the use of BDDs is often sensitive to the used variable ordering, which is sometimes difficult to determine.
For this reason, SAT-based model checking, in particular in the forms of bounded model checking and equivalence checking have recently become very popular. They still benefit from the use of symbolic methods, but tend to be more scalable as they no longer rely on canonical normal forms.
An alternative is the use of explicit state set representations. Clearly, for most real world systems, the state spaces are far too big for a simple explicit representation. Apart from partial order reduction, another important method for coping with the state explosion problem in explicit state model checking, is the so called *on-the-fly* verification. The idea of on-the-fly verification builds upon an observation that in many cases, especially when a system does not satisfy its specification, only a subset of the system states need to be analysed in order to determine whether the system satisfies a given property or not. On-the-fly approaches to model checking (also referred to as local algorithmic approaches) attempt to take advantage of this observation and construct new parts of the state space only if these parts are needed to answer the model checking question.
As mentioned in **Section 2.1.2**, explicit-state automata-theoretic LTL model checking relies on three procedures: the construction of an automaton that represents the negation of the LTL property (negative claim automaton), the construction of the state space, i.e. the product automaton of system and negative claim automata, and the check for the non-emptiness of the language recognised by the product automaton.
An interesting observation is that only those behaviours of the examined system are present in the product automaton graph that are possible in the negative-claim automaton. In other words, by constructing the product automaton graph the system behaviours that are not relevant to the validity of the verified LTL formula are pruned.

As a result, any LTL model checking algorithm that builds upon exploration of the product automaton graph may be considered on-the-fly. We will denote such an algorithm as level 0 on-the-fly algorithm in the classification below.
When the product automaton graph is constructed, an accepting cycle detection algorithm is employed for detection of accepting cycles in the product automaton graph. However, it is not necessary for the algorithm to have the product automaton constructed before it is executed. On the contrary, the execution of the algorithm and the construction of the underlying product automaton graph may interleave in such a way that new states of the product automaton are constructed *on-the-fly*, i.e. when they are needed by the algorithm. If this is the case, the algorithm may terminate due to detection of an accepting cycle before the product automaton graph is fully constructed and all of its states are visited.
Those LTL model checking algorithms that may terminate before the state space is fully constructed are generally considered on-the-fly. If there is an error in the state space (an accepting cycle), an on-the-fly algorithm may terminate in two possible phases: either an error is found before the interleaved generation of the product automaton graph is complete (i.e. before the algorithm detects that there are no new states to be explored), or an error is found after all states of the product automaton have been generated and the algorithm is aware of it. The first type of the termination is henceforward referred to as *early termination* (ET). Note that the awareness of completion of the product automaton construction procedure is important. If the algorithm detects the error by exploring the last state of the product automaton graph before it detects that it was actually the last unexplored state of the graph, we consider this to be an early termination.
We classify "on-the-flyness" of accepting cycle detection algorithms according to the capability of early termination as follows. An algorithm is

- *level 0 on-the-fly algorithm*, if there is a product automaton graph containing an error for which the algorithm will never early terminate.
- *level 1 on-the-fly algorithm*, if for all product automaton graphs containing an error the algorithm may terminate early, but it is not guaranteed to do so.
- *level 2 on-the-fly algorithm*, if for all product automaton graphs containing an error the algorithm is guaranteed to early terminate.

Note that level 0 algorithms are sometimes considered on-the-fly and sometimes not, depending on research community. Since a level 0 algorithm explores full state space of the product automaton graph it may be viewed as if it does not work on-the-fly. However, as explained above, just the fact that the algorithm employs product automaton construction is a good reason for considering the whole procedure of LTL model checking with a level 0 algorithm as an on-the-fly verification process.
To give examples of algorithms with appropriate classification we consider algorithms OWCTY, MAP, and Nested DFS. The OWCTY algorithm is a level 0 algorithm, the MAP algorithm is a level 1 algorithm and Nested DFS is a level 2 algorithm.

As with state space exploration algorithms, the efficiency of the on-the-flyness of the algorithm may also be improved by other techniques. It might be the case that even the level 2 on-the-fly algorithm fails to discover an error, if the examined state space is large enough to exhaust system memory before an error is found. This issue has been addressed by methods of directed model checking [60–62], which combines model-checking with heuristic search. The heuristic guides the search process to quickly find a property violation so that the number of explored states is small. It is worthwhile to note that this approach can be extended with directed search as well.

2.6.8 OWCTY On The Fly

The idea of propagating one accepting predecessor along all newly discovered edges is at heart of a heuristic extension of OWCTY [15]. If the propagated accepting state is propagated into itself, an accepting cycle is discovered and the computation is terminated. Like with the MAP algorithm, an accepting state to be propagated is selected as a maximal accepting state among all accepting states visited by the traversal algorithm on a path from the initial state of the graph to the currently expanded state. Since the INITIALISE phase of OWCTY needs to explore full state space, we can employ it to perform limited accepting cycle detection using maximal accepting state propagation. Unlike the MAP algorithm, we however avoid any re-propagation to keep the INITIALISE phase complexity linear in the size of the graph. This means that some accepting cycles that would otherwise be discovered (i.e. with relaxation, or re-propagation, enabled) may be now missed. In particular, there are three general reasons for not discovering an accepting cycle with the heuristic.

---

ex. 2.1 There are three main scenarios where the MAP heuristic fails to discover an accepting cycle. a) Maximal accepting predecessor is out of the cycle. b) There is no fresh path back to the maximal accepting state. c) Wrong order of propagation, $C \to D$ is explored before $B \to D$, hence, C is propagated from D.
In case (a), the maximum accepting predecessor of the cycle does not lie on the cycle itself, preventing propagation back to itself. In case (b), the maximum accepting predecessor value does not reach the originating state because there is no fresh path (a path made of yet unvisited states) that could reach it. Finally, the maximum accepting predecessor value can fail to reach the originating state due to wrong propagation order – this is case (c).

When the algorithm encounters an accepting state that is being propagated, it terminates early, producing a counterexample. On the other hand, if the INITIALISE phase (i.e. the first reachability) of OWCTY fails to notice an accepting cycle, the rest of the original OWCTY algorithm is executed. Either the algorithm finds an accepting cycle (and again, produce a counterexample), or it proves that there are no accepting cycles in the graph.

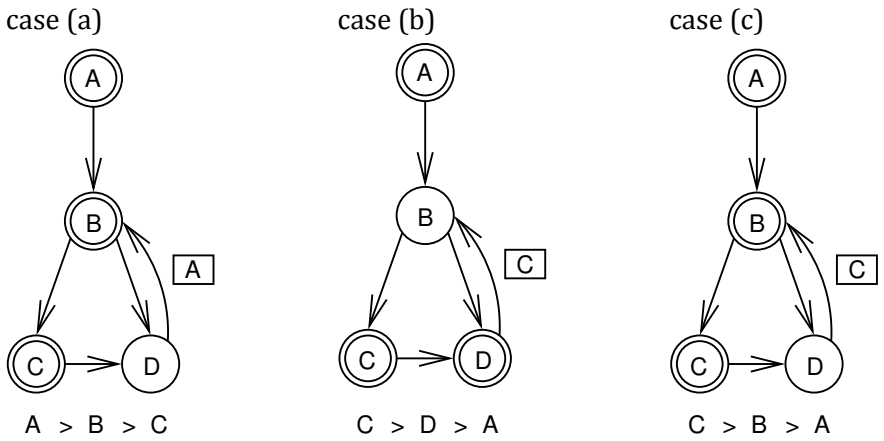


fig. 2.1 Three scenarios where no accepting cycle will be discovered using accepting state propagation.

2.7 State Space Reductions

For a predominantly unstructured model representation, like the one comprised by an assembly-level language, *reduction* techniques appear to be the most suitable choice for obtaining smaller, more manageable state spaces. While using abstractions may be an option in the future, these are normally applied to much more structured model representations (i.e. on the level of program source code, or equivalent). We anticipate significant challenges in implementing useful abstractions over unstructured, assembly-level programs.

Two main categories of state space reduction techniques are applicable in our case: partial order reduction [77, 78, 127, 129 and 149] (POR for short) and symmetry reduction [47 and 66]. The former is extremely useful to reduce state spaces of *parallel* programs, by avoiding exploration of superfluous execution interleavings. On the other hand, *symmetry* reduction is successful whenever a number of symmetric configurations of a particular system are equivalent with respect to the property of interest. In the case of software, many different dynamic memory (heap) configurations can be considered equivalent (symmetric), whenever they only differ in ordering (but not content, after adjusting the pointers to other heap areas).

While the most common approach to symmetry reduction is independent of exploration order, and therefore parallelism, since it only uses state-local information, this is not the case with reductions based on partial order. In fact, the most efficient, most widely used implementation of POR in LTL model checking is based on depth-first postorder, which is inherently sequential as outlined above and a comparable parallel implementation has been elusive until recently [14].

Moreover, when applying partial order reduction to a state space of a parallel LLVM assembly program, we expect to see many non-branched chains of states, due to very fine granularity of interleaving and many of the transitions being completely invisible – in those cases, partial order reduction will only pick a single interleaving of all the

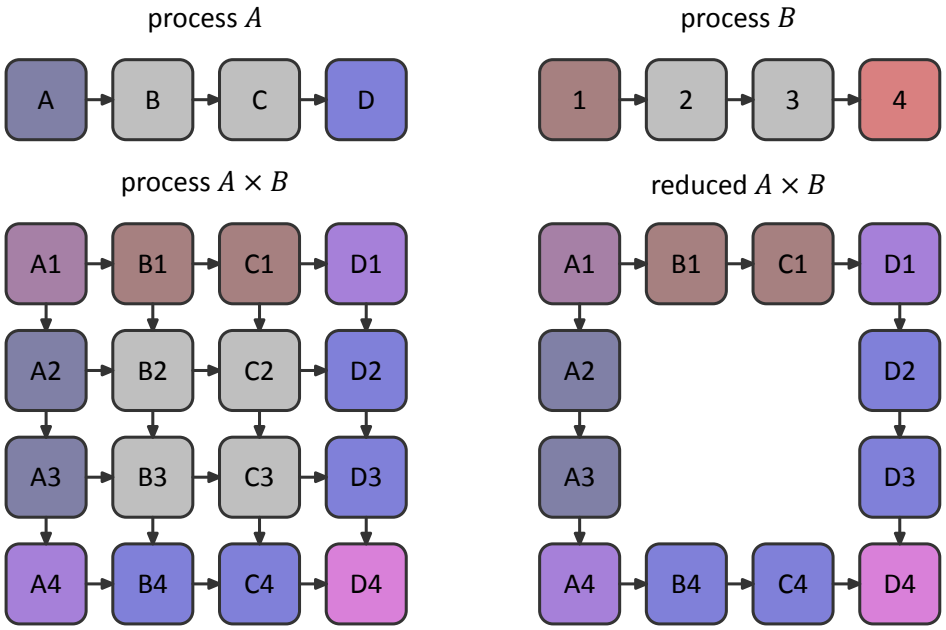


fig. 2.2 Behaviour of POR in a system where invisible actions are prevalent.

invisible instructions schedulable between the nearest visible actions. This is illustrated in **Figure 2.2**. Therefore, it is desirable to combine this partial order reduction with a suitable *path reduction* [100 and 157], that will eliminate the redundant intermediate states. The desirable overall outcome is shown in **Figure 2.3**.
A partial order technique of special interest for software model checking is based on the notion of *transactions*, where a transaction is defined as a block that is, from the point of view of any other thread of execution, atomic [74].

2.7.1 Partial Order Reduction

The Partial Order Reduction technique has been intensively studied as a leading technique to fight the state explosion problem in explicit model checking. As a result, a number of improvements and variants of the technique have been developed and successfully integrated in verification tools. These results are mutually exclusive in many cases and their usability depends on the target domain of application. In particular, there are subclasses of properties to be verified for the system under consideration, for which the formal requirements on *ample* sets may be safely weakened, hence different reduction algorithms applied.

For example, to prove deadlock freedom, the reduced structure does not have to fulfil the **C3** property at all. Similarly, if we check the system for a safety property, such as an assertion violation, it is satisfactory for the states on a cycle in the reduced structure to be able to reach at least one fully expanded (not necessarily immediate) successor state [30]. In the following we will focus on various strategies to deal with **C3** proviso that have been introduced in the literature so far. We will particularly discuss their applicability to the distributed-memory computing.

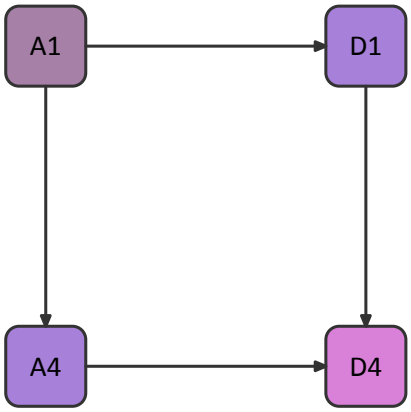


fig. 2.3 Combined reduction.

2.7.1.1 Static

Static Partial Order Reduction [107] builds upon the fact that the system under consideration is an asynchronous product of individual system components. Since every cycle in the system graph projects to a cycle in each of the components, it is possible to a-priori construct a set of states that cover every possible cycle in the system graph. Whenever a state of the reduced structure is a member of such a covering set, it is fully expanded. The static partial order reduction technique is compatible with distributed-memory computation; however, it is generally considered to be less effective than dynamic approaches listed below.

2.7.1.2 Dynamic

In the dynamic Partial Order Reduction approach, the decision about the full expansion of a state is done for the state when it is processed by the exploration algorithm. There are several nuances of the cycle proviso (condition **C3**) that depend on whether the reduced structure is used for verification of safety or liveness properties, or whether the exploration algorithm follows a particular search order (depth-first, breadth-first, etc.).
The classical, stack-based cycle detection proviso is connected with a depth-first traversal algorithm. The depth-first search algorithm maintains a stack of states on the path from the initial state of the graph to the state currently being processed. If the currently processed state has a direct successor that is on the stack, there is a cycle in the reduced structure. In the case of verification of liveness properties, such a situation requires that the currently processed state must be fully expanded. However, this is not the case when verifying safety properties, where the full expansion of the currently processed state may be safely avoided if there is at least one direct successor of the

state that is outside the stack [90]. In other words, a state is fully expanded if all its successors are in the stack.
If for whatever reason the algorithm for exploration of the reduced structure does not follow a depth-first visiting strategy, it cannot maintain the search stack, hence, cannot apply the stack proviso. In general, whenever a graph traversal algorithm discovers a transition leading to an already visited state, there is a potential risk that this transition closes a cycle in the reduced structure. A conservative approach therefore is to fully expand all states that have successors lying in the visited portion of the graph.
Unfortunately, even this conservative check is not free of issues in distributed memory setting – checking whether a state is visited will cost two messages and, what is worse, the successor generation will need to wait for the answer, introducing extra synchronisation (and therefore delays) into the system. In shared-memory setting, the problem is easier to resolve (the visited check can be implemented more efficiently).

2.7.1.3 Parallel

A comprehensive survey of existing techniques for parallel POR and an exhaustive experimental evaluation is available in [121]. The work also introduces a number of novel techniques for distributed-memory reductions, although they are often rather difficult to implement and often rely on successor locality to a workstation. The latter requirement, that is, a special treatment of so-called "cross" transitions, proves to be increasingly problematic with higher numbers of parallel workers – quickly, all transitions become „cross" and the often more complicated and suboptimal treatment is required for all transitions of the system. In contrast, our proposed **C3** check is independent of state space distribution and therefore a highly scattered state distribution (which is common with hash-based state distribution) does not pose a problem for the check. In a similar fashion, the check proposed in [34] relies on the ability to do a local depth-first search, which in turn relies on availability of local (i.e. non-cross) transitions. Additional heuristic is proposed, that improves handling of cross transitions at the cost of visiting any given state multiple times (at most once for each worker involved in the computation). This unfortunately still translates to a high penalty for cross transitions.
A different approach to POR has been proposed in [124] – an algorithm that does not rely on **C3** at all, and instead relies on following singleton ample sets while this is possible and fully expanding otherwise. The algorithm shows promise, although it introduces complications into generation of successors and for distributed computation. It is also less general than the usual POR approach which is not restricted to singleton ample sets, even though authors claim it often outperforms the traditional approach in practice. The ramifications of a parallel, distributed implementation of this algorithm are currently not known and are a subject of future work.
For the case of reachability (i.e. verification of safety properties) on shared-memory systems, a heuristic [89] has been proposed that is usable with a multi-core extension of the SPIN model checker. The check assumes usage of a so-called stack-slicing algorithm (this is the reason the heuristic requires a shared memory environment), proposed

in [88]. The heuristic itself, similar to the above-mentioned distributed algorithms, treats cross transitions (as represented by boundary states, in SPIN terminology) specially – in this case, however, this is less problematic, since the partitioning of the state space using the stack-slicing algorithm is not static and therefore the proportion of border states in the state space is easier to control.

2.7.1.4 Transactions

An alternative approach to reducing state spaces of multi-threaded programs is to find sections of code that are atomic with regards to accessing a particular variable. To a certain degree, this approach assumes mutex-based protections on shared variables and an equality relationship on variables[14], and hence is limited to a particular class of programs. The idea is to infer predicates that guarantee exclusive access to a particular memory location, based on the status of locks held by the accessing thread. The main property required of the predicates $E(t, x)$ and $E(u, x)$ for t, u threads and x a variable is that these can *never* hold simultaneously for $t \neq u$. The paper [74] presents a procedure to find such predicates automatically for a class of multi-threaded programs and how to statically infer transactions based on those predicates. Since a transaction has the property that steps within the transaction cannot be observed by other threads, the model checker can hence treat it as an atomic section of code, reducing the number of interleavings, and consequently, the size of the state space.

2.7.2 Symmetry Reduction

Symmetry reduction builds on the idea that certain states of the system are symmetric to each other in some respect. There are basically 2 criteria for symmetry: clearly, the states need to be symmetric with regards to the property, and secondly, their successors need to be likewise symmetric. If these conditions are met, we can reduce the state space by only considering one state out of a set of symmetric states.
The successor condition ensures that all the states that we remove from the state space by not considering symmetric states will be symmetric to *some* generated states.
There are two major classes of symmetries with the requisite properties: first, explicit symmetry via a *symmetric set* construction – the model explicitly specifies that a particular set of values (threads, processes, etc.) is symmetric – it is a "true" set with no ordering, and hence the order of elements in the set cannot influence the outcome of any operation. This is enforced at a type level; the operations available on the *symmetric set* type do not expose any property that could change based on the order of elements in the set.
The second class of symmetries of this kind are *heap symmetries* or more generally *graph symmetries*. The idea here is that there are many ways to lay out a particular

[14] In real C and C++ programs, such equality is not available, and can be, at best, approximated using alias analysis. See also **Section 7.6**.

graph (heap[15]) in a linear address space. Nevertheless, as long as the program is disallowed from making numeric comparisons of pointers to different heap objects, different layouts of the same graph are indistinguishable, and as such conform to the definition of symmetry above.

2.8 Linear Temporal Logic

We will interpret LTL formulae on the structure known as a Kripke structure [104], i.e. a triple $(s_0, S, \rightarrow)$ where $s_0 \in S$ and $\rightarrow \subseteq S \times S$: with s_0 representing the initial state of the system, $\rightarrow$ representing the transitions of the system and S comprising all the possible states. Additionally, we define L a set of labels, and $\psi : S \mapsto 2^L$ a labelling function.

def. 2.2 The syntax of an LTL formula is defined by the following recursive equation:

$$\varphi := true \mid atom \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathcal{X}\varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

■

A *path* π in a Kripke structure is an *infinite* sequence of states, $s_1 s_2 \ldots$, such that for every i, $s_i \rightarrow s_{i+1}$. We use π^n as a succinct way to express the suffix of a path that has first n states removed (i.e. $\pi^3 = s_4 s_5 \ldots$).

def. 2.3 A path $\pi = s_1 s_2 \ldots$ with a labelling function ψ satisfy an LTL formula φ, that is, $(\pi, \psi) \vDash \varphi$ iff:

$$\begin{aligned}
(\pi, \psi) &\vDash \quad true \quad \Leftrightarrow true \\
(\pi, \psi) &\vDash \quad atom \quad \Leftrightarrow atom \in \psi(s_1) \\
(\pi, \psi) &\vDash \quad \neg\varphi \quad \Leftrightarrow (\pi, \psi) \nvDash \varphi \\
(\pi, \psi) &\vDash \varphi_1 \wedge \varphi_2 \Leftrightarrow (\pi, \psi) \vDash \varphi_1 \wedge (\pi, \psi) \vDash \varphi_2 \\
(\pi, \psi) &\vDash \quad \mathcal{X}\varphi \quad \Leftrightarrow (\pi^1, \psi) \vDash \varphi \\
(\pi, \psi) &\vDash \varphi_1 \mathcal{U} \varphi_2 \Leftrightarrow \exists i. (\forall j < i. (\pi^j, \psi) \vDash \varphi_1) \wedge (\pi^j, \psi) \vDash \varphi_2
\end{aligned}$$

A model (the Kripke structure with an associated labelling function) satisfies an LTL formula iff all possible paths in the structure starting at s_0 satisfy the formula. [16] ■

[15] In the sense of dynamic memory with pointers.

[16] This "builtin" universal quantifier in the semantics of LTL has interesting implications for behaviour of negation. In LTL, the law of the excluded middle does not hold: there are pairs of an LTL formula φ_i and a structure S such that $S \nvDash \varphi_i \wedge S \nvDash \neg\varphi_i$.

def. 2.4 A number of derived, more intuitive operators (shortcuts) is usually provided as part of LTL:

$$\begin{aligned} false &\stackrel{\text{df}}{=} \neg true \\ \varphi_1 \vee \varphi_2 &\stackrel{\text{df}}{=} \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\ \varphi_1 \Rightarrow \varphi_2 &\stackrel{\text{df}}{=} \neg\varphi_1 \vee \varphi_2 \\ \mathcal{F}\varphi &\stackrel{\text{df}}{=} true\, \mathcal{U}\varphi \\ \mathcal{G}\varphi &\stackrel{\text{df}}{=} \neg\mathcal{F}\neg\varphi \end{aligned}$$

Intuitively, a path satisfies $\mathcal{X}\varphi$ if the "next" state of the path satisfies φ, $\mathcal{F}\varphi$ whenever there is a state on the path that satisfies φ (i.e. somewhere in the future), $\mathcal{G}\varphi$ whenever *all* states of a path satisfy φ.[17] Of course, operators can be nested arbitrarily, with the most common nestings being $\mathcal{F}\mathcal{G}\varphi$ meaning that from some point in future, φ will hold continuously, $\mathcal{G}\mathcal{F}\varphi$ meaning φ holds infinitely often (i.e. φ holds at least once, and whenever it holds, it'll hold again in the future) and $\mathcal{G}(\varphi_1 \Longrightarrow \mathcal{F}\varphi_2)$, meaning that whenever φ_1 "happens", φ_2 will eventually follow.
Apart from LTL itself as a specification language, since we employ the automata-based approach to LTL model checking [150], we can, alternatively, specify properties in terms of Büchi automata, which may be more desirable in some cases (properties stated in LTL are more declarative in nature, while properties expressed through automata are more procedural: a trait that may make automata-based specifications more palatable to engineers without extensive formal method training).

2.8.1 Atomic Propositions

One of the challenges in applying LTL to model checking programs lies in formulating suitable semantics for the atomic propositions in the LTL formulae. In specialised modelling formalisms, it is usually the case that atomic propositions are allowed to be arbitrary expressions of the modelling language, referring to any state variables present in the system: the number and locations of such variables tend to be fixed in the system. Under these assumptions, assigning atomic propositions (labels) to states of the system is trivial: whenever the associated expression evaluates to *true* in a given system state, the atomic proposition holds in that state.
In general-purpose programming languages, however, state variables are often lexically scoped. Referring to such variables in an LTL formula may be desirable, but it is not

[17] The formulae of the form $\mathcal{G}atom$ exactly correspond to "reachability" properties mentioned earlier: the absence of deadlock and the absence of assertion violations.

entirely clear how this should be done.[18] To our best know, this problem has not been addressed in the literature, a fact that is attributable to the general neglect of LTL in relation to model checking of programs. In **Section 5.2**, we will explore a new approach to this problem.

2.8.2 Related Logics

LTL does not live in a vacuum, quite to the contrary, it is part of a wider family of modal and temporal logics. The closest relatives of LTL are CTL [46] (computational tree logic) and a superset of both, CTL* [65]. CTL* is in turn a subset of modal μ-calculus [103]. All of these logics have been targeted in automated model checkers, although CTL and LTL enjoy the widest use, in model checking synchronous and asynchronous systems respectively.

A different approach to specification of temporal properties is taken by TLA [112] – Temporal Logic of Actions. Like LTL, TLA formulas describe behaviours, or system runs. While LTL is usually either fully state-based (atomic propositions refer to states) or fully action based (atomic propositions refer to transition labels), TLA is intrinsically mixed. However, actions in TLA are always expressed in terms of states: it contains a special language for describing actions in terms of *before–after* relationship between variables. An action in TLA might look something like $y' = y + 1$, meaning that if y is a variable value in state s, value y' in the successor state s' is y incremented by one. Additionally, TLA formulas can be subscripted by an expression – $[A]_f$ – meaning that the formula holds when either A holds or when the value of f remains unchanged.

One of TLA's advantages over LTL is that it is intrinsically stutter-invariant, and as such, specifications written in TLA are more composable than comparable LTL specifications. This is achieved by requiring that the operand of $\mathcal{G}$ is always of the form $[A]_f$ for some f, which makes the formula true whenever $f' = f$. As such, all variables that could cause the evaluation to become "blocked" must be explicitly listed – specifically, if only variables not mentioned by a particular subformula change, this cannot cause a $\mathcal{G}$ clause to fail. Since stutter-invariant formulas are also intrinsically unfair, TLA contains syntactic shortcuts for expressing both weak ($\mathcal{FG}enabled \Rightarrow \mathcal{GF}taken$) and strong fairness ($\mathcal{GF}enabled \Rightarrow \mathcal{GF}taken$), which share the "subscripting requirement" of the $\mathcal{G}$ operator.

As such, formulas in TLA can be combined naturally – unlike in LTL, if process P satisfies specification π and process R satisfies specification ρ (and π, ρ use distinct variables), process $P||Q$ satisfies specification $\pi \wedge \rho$ – in LTL, π could all too easily fail during a step done by Q.

[18] In LTL extensions to Java PathFinder (JPF), a variable is specified using its name and its enclosing function. From the documentation, it is not clear what happens when a variable is shadowed in an anonymous subscope of the function (i.e. the atomic proposition could refer to the shadowed name, as well as to the original, temporarily inaccessible variable, and in fact, both interpretations are sensible in different contexts). Moreover, neither of the two existing LTL modules for JPF document the semantics of an atomic proposition that mentions variables that are currently out of scope.

Even though TLA is more-or-less equivalent to LTL without the $\mathcal{X}$ operator[19], we still need to consider the fact that large part of TLA's power stems from its ability to concisely describe actions, a feature entirely missing from LTL. Overall, TLA is more "user-friendly" than LTL in most scenarios; unfortunately, it is only rarely used in the context of model checking. This might be attributed, among other things, to the fact that it is substantially more complicated to implement.
LTL is also related to monadic first-order logic – namely, FO[<] has the same expressive power as LTL [131] (that is, they both recognize the class of star-free ω-regular languages). As such, these logics cannot fully characterise all stutter-free ω-regular languages. The language μTL($\mathcal{U}$) is a fixpoint extension of $LTL_{-\mathcal{X}}$ (which is the same stutter-invariant fragment of LTL as LTL ($\mathcal{U}$)), and covers all stutter-invariant ω-regular languages. An even more powerful formalism is the modal μ-calculus (a strict superset of μTL) which also describes branching time (although it is not stutter-invariant).

[19] L. Lamport, the inventor of TLA, writes in [112]: "TLA can express all formulas invariant under stuttering that Manna and Pnueli's logic can. However, their logic can also express formulas that are not invariant under stuttering. Such formulas yield specifications that cannot be refined. Although all TLA formulas are expressible in Manna and Pnueli's logic, there is no simple translation from TLA to their logic because its quantification operator is not invariant under stuttering. "

3 Parallel Search Implementation

There are multiple approaches to implementation of parallel algorithms, most notably differing in the treatment of memory: whether each parallel process has its own private working memory and all communication is implemented as message passing, or whether the parallel processes access and modify a single shared data structure. This distinction is partially dictated by hardware: it is simpler and cheaper to design hardware with local memory and message passing (even low-latency, high-bandwidth message passing) than it is to design systems with uniform memory access across all processors.

However, small-scale (semi-)uniform memory architectures are widespread today, with almost all computers sporting multi-core CPUs. It is therefore desirable to "scale down" parallel systems originally designed for clusters in a way that makes them run efficiently on modern "small scale" parallel hardware. Multiple stages of such down-scaling are possible: software written for MPI-based [118] clusters can be directly executed on SMP machines using loopback interfaces, or even using an implementation of MPI optimised for passing messages through shared memory buffers. Such an implementation, however, is largely inefficient since it still obeys many of the limitations dictated by distributed memory hardware, which are no longer applicable on a shared memory platform.

A slightly more involved approach is to replace MPI-based message passing with data structures optimised for shared memory, most importantly use of lock-free, wait-free IPC queues for passing data, and to a lesser degree, use of efficient shared memory barriers. This approach was pioneered in [9], providing significant scalability improvements. We will briefly discuss the data structures that are used to this effect in **Section 3.2** and **Section 3.3**.

Finally, further scalability improvements can be achieved by leveraging the uniformness of memory access to the largest extent possible, i.e. by fully sharing the data structures used by the algorithm among processes. In a typical graph exploration application, the two most important data structures are a queue and a set, the latter usually implemented as a hash table. We will discuss these structures in **Section 3.4** and **Section 3.5**.

While contemporary SMP systems provide vast processing power, it cannot rival that of large clusters, counting in hundreds or even thousands of nodes. However, modern clusters use a hybrid architecture, with each node of the cluster becoming a small parallel machine in its own right. Algorithms that offer the ability to run in distributed memory can be, using the techniques outlined above, transformed into efficient shared-memory algorithms. Fortunately, this transformation does not change the algorithms themselves in a way that would prevent their use with distributed memory: in fact, the execution of these algorithms can be arranged in a way that mimics the architecture of modern clusters, using a shared-memory implementation within each cluster node, but

falling back to distributed-memory techniques for communication across node boundaries. This way, execution of the algorithm can use the available hardware resources most efficiently.
Additionally, systems with shared memory pose one additional challenge for implementation of parallel algorithms. While memory allocation and deallocation is usually taken for granted in sequential algorithms without giving it a second thought, the situation is not as simple in the parallel world: hardware provides a flat memory space, and in order to make efficient use of this flat space, programs use elaborate data structures to organise it. However, these data structures are as much subject to concurrent access as any other data structures explicitly used in a parallel algorithm, even though their use is usually implicit in an algorithm's design. Hence, the design of these data structures can be crucial for practical performance of such algorithms, and this is especially true of allocation-intensive workloads (and graph exploration in model checking is in fact a very allocation-intensive problem). Hence, in **Section 3.7**, we will discuss the problem of memory allocation in a parallel shared memory program.

3.1 Data Structure & Algorithm Design

In this section, we will discuss the considerations that go into designing a data structure or an algorithm, with special emphasis on parallelism and concurrency.

3.1.1 Hardware Limitations

While multi-core and SMP systems present a single very large address space, they also exhibit a deep memory hierarchy, with many levels of cache. Some of this cache is shared by multiple cores, some is private to a particular core. This translates into a complex memory layout. To further complicate matters, multi-CPU computers nowadays often use non-uniform access architecture even for the main memory: different parts of RAM have different latency towards different cores. Most of this complexity is implicitly hidden by the architecture, but performance-wise, this abstraction is necessarily leaky. Finally, the gap between the first and the last rungs of the hierarchy is huge: this means that compact data structures often vastly outperform asymptotically equivalent, but sparse structures. Due to cache organisation constraints, memory cells that live close to each other are usually fetched and flushed together, as part of a single "cache line". They are also synchronised together between core-private caches. A modern data structure should therefore strive to reduce to an absolute minimum the number of cache lines it needs to access in order to perform a particular operation. When concurrency is involved, there is a strong preference to have threads use non-overlapping sets of cache-line-sized chunks of memory, especially in hot code paths.

3.1.2 Benchmarking & Performance Evaluation

In previous section, we have laid out the guiding principles in implementing scalable data structures for concurrent use. However, such considerations alone cannot guarantee good performance, or scalability. We need to be able to compare design variants, as well as implementation trade-offs and their impact on performance. In turn, we need a reliable and comprehensive way to measure performance: this comes down to finding *what* to measure and *how* to measure it. Both are surprisingly tricky: in order to identify what to measure, we need to have a fairly good idea on how is the data structure or algorithm going to be used in the bigger picture.
We will discuss the *what* individually for each set of benchmarks. There are, however, many common factors in deciding *how* to measure and we will detail them here.
The main problem with computer benchmarks is *noise*: while modern CPUs posses high-precision timers which have no impact on runtime, modern operating systems are, without exceptions, multitasking. This multitasking is a major source of measurement error. While in theory, it would be possible to create an environment with negligible noise – either by constructing a special-purpose operating system, or substantially constraining the running environment, this would be a huge investment. Moreover, we can, at best, hope to reduce the errors in our measurement, but we can hardly eliminate them entirely.
Henceforth, we accept the existence of measurement errors as a fact of life (and there seems to be little choice in the matter). However, since all kinds of measurements – not only benchmarking – are subject to error, a comprehensive toolkit of statistical methods to counteract those errors already exists. While many methods are available, benchmark results have a certain peculiarity: the distribution of the times we measure is decidedly non-normal (in the statistical sense). In fact, it's often hard to predict what the distribution is, and this may even depend on circumstances of the measurement (in other words, the same benchmark may produce markedly different distributions on different days).
One way to counteract these effects is to choose a robust estimator, such as median, instead of the more common mean. However, since we only posses finite resources, we can only obtain limited samples – and even a robust estimator is bound to fluctuate unless the sample is very large. Ideally, we would be able to understand how good our estimate is. If our data was normally distributed (which we know is, sadly, not the case) we could simply compute the standard deviation and base a confidence interval for our estimator on that. However, since we need a computer for running the benchmarks anyway, we can turn to bootstrapping: a distribution-independent, albeit numerically intensive method for computing confidence intervals.
Bootstrapping works as follows: since the sample we have is the best (and only) data about the phenomenon we are measuring (i.e. the hypothetical population of all possible data points), we need to derive the properties of the sample from the sample itself. The way to do this is to take samples from the sample: this is called resampling, and is done with the same sample size, but *with replacement*. This means we draw data points

from our sample at random, without eliminating them – a single data-point will likely end up in the resample many times, while others will be absent.[20] We can compute our estimator of interest on this resample, and obtain a different value – a value that is different from the one for the original sample, yet very likely to be just as realistic. If we repeat this resampling and estimation many times, we obtain a large number of estimator values that are more or less valid for the original phenomenon – a sample in its own right.

Now if we were to take an entirely new sample of the original phenomenon, the value of our chosen estimator for this sample would be overwhelmingly likely to fall into the range of our resampled (bootstrapped) estimator sample. This will remain true even if we only take the central 95% of the resampled estimator values into account. In other words, the 2.5th and 97.5th percentile of the bootsrapped sample constitute a (very nearly) 95% confidence interval for our estimator of choice. While a single outlier can substantially affect the mean value of a sample, many of the resamples will lack this particular outlier, and as such, a value of the mean that excludes this particular outlier will still fall within our bootstrap-computed confidence interval.

While bootstrapping gives us a good method to compute reliable confidence intervals on population estimators, it doesn't do anything to make those confidence intervals tighter. Given a sample with high variance, there are basically two ways to obtain a tighter confidence interval: measure more data points, or eliminate obvious outliers. While a bigger sample is always better, we are constrained by resources: each data point comes at a cost. As such, we need to strike a balance. For measurements in this thesis, we have removed outliers that fell more than 3 times the interquartile range (the distance from the 25th to the 75th percentile) of the sample from the mean, but only if the sample size was at least 50 measurements, and only if the confidence interval was otherwise more than 5% of the mean.

Finally, most of our benchmarks measure time as it scales with some parameter (or two). Not only is the measurement of a single data point affected by resource constraints, but so is the sampling of these parameters. Ideally, we would be able to measure every sensible parameter value separately – however, this would in many cases cover millions of data points, which is entirely impractical. We have chosen to measure (at most) a few dozen data points for each benchmark, spread out evenly over the range of the parameter[21] and use cubic spline interpolation to fill in the gaps.

In all benchmark plots in this thesis, we use a line plot for the mean value of the measurement sample, with black dots on the line representing the actual measurements, the line being an interpolation of those. The shaded area in the same colour then represents the 95% confidence interval obtained via bootstrapping.

[20] Bootstrapping is closely related to a technique called the jackknife – where the stochastic approach of bootstrap is replaced by a deterministic elimination of one data point at a time.

[21] I.e. linearly for linear parameters and exponentially for exponential parameters

3.1.3 Verification

In order to ensure that the data structures we use work as expected, we have used DIVINE to check some of their basic properties. The properties are expressed as small C++ programs – basically what a programmer would normally call a unit test. They are usually parametric, with the parameters governing the size and parameters of the data structure as well as the way it is used. A property of a queue, for instance, may specify the maximum length of the queue, if the queue is shared a number of threads that access it, or the number of distinct elements to insert into the queue.
Clearly, the parameter space of such a property is infinite, and admittedly, even for fairly small values the verification problem becomes very large. Nevertheless, most bugs happen in boundary conditions, and these are identical for all parameter instantiations upwards of some structure-specific minimum. The amount of "fluff" data in the data structure is largely irrelevant in systematic exploration.[22]
The second limitation is that we can only currently verify the code under the assumption of sequential consistency. At first sight, this may seem like a severe limitation – on a closer look, though, it turns out that vast majority of relevant memory access is already tagged as sequentially consistent using appropriate `std::atomic` interfaces (this translates to appropriate architecture-specific memory access instructions that guarantee sequential consistency on the value itself, as well as working as a memory fence for other nearby memory accesses). In this light, the limitation is not quite fatal, although of course it would be preferable to obtain verification results under a relaxed memory model.

3.2 IPC Queues

Normally, an implementation based on message passing will operate on the premise of seeing an entirely distinct address space in each process. This means that all data that needs to be shared with another process must be packed in its entirety into a message and shipped to the other process somehow. If this data is bulky, this packing represents significant overhead. With a little care on the side of the receiver, we can immediately improve on this by only sending pointers to data residing in memory shared by the processes. The additional requirement on the receiving side is that a copy of the data must be made before any modifications are done to it. However, often the data is entirely read-only and the communication can become zero-copy.
However, we still need to pass around the pointers to the bulk of data between processes. One way to go about this would be to obtain a shared memory area mapped at the same address into multiple processes and use MPI to communicate the pointers. However, this approach breaks down if we use multi-threading instead of multi-processing in our application: MPI is ill-suited for passing messages between threads. Also, the latency

[22] If we use random sampling (as in, for example, testing multi-threaded code), bigger ("fluffier") data points contribute to our chances of discovering a problem. This intuition somewhat skews our perception of verification results.

of MPI is, compared with latency of RAM, still unreasonably high. Instead, we can take advantage of shared memory to move the pointers from one thread to another by using an appropriate data structure.
In order to minimise latency and overhead, we should not need to take any locks when writing to, or reading from, this data structure. It should operate in FIFO order, just like MPI messages: data written first by the sender should be the first seen by the reader. Hence, the data structure we need is a lock-free queue with one writer and one reader thread. There are many ways to implement such a data structure – however, to make most of the hardware, we need to ensure, in accordance with the considerations laid out in **Section 3.1.1**, that it is cache-efficient [67] as well, and that it reduces the amount of read-write alterations on a single cache line [147] to the necessary minimum.
Our design uses a lock-free queue for passing pointers between threads, implemented with good memory locality in mind: instead of using a linked list of pointers – which can be implemented in a lock-free fashion, we instead use a linked list of blocks, each block containing a constant number of items. This not only improves cache locality by using contiguous blocks of memory for related data, it also reduces memory overhead significantly (the savings asymptotically approach 50% as the block size increases). Each block contains a pair of indices, one for indicating the reading, and one for writing position – ensuring that reads and writes to the same block (when the queue is nearly empty) do not contend on a single memory location.
The enqueue and dequeue operations are described in more detail **Algorithm 3.1** and **Algorithm 3.2**, respectively.

---

alg. 3.1 Unlike with high-level algorithms, we will use C++ fragments to give a formal presentation of the algorithm for manipulating the IPC queue. When pushing new elements, the algorithm first needs to check whether the currently active queue node has enough space in it, and if no, create a new node to be pushed onto the linked list:

```
Node *t;
if ( tail->write == tail->buffer + NodeSize )
    t = new Node();
else
    t = tail;
```

Now that we selected the target node (in case of a new node, this node is *not* yet part of the linked list, since it must not be chained in while it is empty. The `__sync_synchronize` call inserts a memory fence, barring both the compiler and the CPU from re-ordering the update of the `write` pointer with the update of the buffer (buffers represent blocks of data in the queue) inside the node. I.e., the write pointer must only be updated when the value in the buffer was already written, so that another core does not see the new `write` value and read the content of the buffer before the current core finishes writing.

```
*t->write = x;
__sync_synchronize();
++ t->write;
```

Finally, the enqueue algorithm, in case it has created an entirely new node, needs to update the top-level linked list of nodes. This entails first updating the `next` pointer of the current `tail` and then updating the `tail` pointer itself to the new node. Again, a memory fence prevents incorrect re-ordering of memory accesses. The `tail` pointer needs to be updated last, because it is used in the emptiness check. If the writes were reversed, another thread may conclude there are more nodes in the queue (because `head != tail`) but access a `NULL` pointer still stored in the `next` pointer of the previous `tail` (in cases where it is equal to `head`).

```
if ( tail != t ) {
    tail->next = t;
    __sync_synchronize();
    tail = t;
}
```

Besides inserting elements, the queue needs to support reading and removing elements, a process that happens in another thread, and as such is asynchronous (concurrent) with the algorithm above. Of course, the threads may be running on the same core, or on different cores. In both cases, the code needs to be very careful about operation ordering.

---

alg. 3.2 While the enqueue algorithm operates on the tail of the linked list, the reverse is true of dequeuing. Fetching a value from the queue is done in multiple steps, and is therefore slightly more complicated than inserting values is. First, we need to be able to check whether the queue has any values in it, before attempting to fetch one:

```
bool empty() {
    return head == tail && head->read >= head->write;
}
```

As outlined in the description of **Algorithm 3.1**, we require that the `tail` pointer is updated last, so the emptiness check errs on the safe side: "falsely" reporting that the queue is empty even if it is not. Since the reader and the writer are concurrent to each other, this situation must be handled by the code, whether the emptiness check of the queue is exact or not: a new value might arrive into the queue right after the emptiness check has finished executing in the reader thread. Claiming a non-empty queue while it was empty, however, would be fatal for the rest of the code, which would then inadvertently manipulate invalid pointers.

The next piece of fetching an item is actually reading the value at the front of the queue. This would be trivial, was it not for an edge case where a full block was emptied but not dropped from the linked list yet. In that case, the node needs to be discarded. The invariant that the queue is not entirely empty at this point allows us to assume that `head->next` is a valid pointer.

```
T &front() {
    if ( head->read == head->buffer + NodeSize )
        head = head->next;
    return *head->read;
}
```

Finally, after we have read it, we need to be able to remove the value at the front of the queue. Again, we need to take care of the same edge case, since the data structure allows reading and removing values independently: for example when an algorithm decides that it won't need to examine further graph edges in a particular round, it can remove values from the IPC queue without examining them. Finally, if the active `head` node has become empty, it is removed from the queue. Please note that the details of memory management have been elided from the code to improve readability.

```
void pop() {
    ++ head->read;
    if ( head->read == head->buffer + NodeSize && head != tail )
        head = head->next;
    if ( head != tail && head->read > head->buffer + NodeSize ) {
        head = head->next;
        pop();
    }
}
```

3.2.1 Verification

For verifying that the IPC queue behaves as expected, we have used the following property: for two threads (since an IPC queue always operates between two threads, hence in this case the number of threads is not a parameter of the property), one a writer and one a reader, the values inserted in the writer are the same as seen by the reader. The property has 3 parameters, B for block size of the queue (in number of items), L the maximum length of the queue and N, the number of distinct elements that are inserted cyclically into the queue by the writer (and checked by the reader). There are 2 global variables in the test:

```
using Q = brick::shmem::Fifo< int, B >;
Q *q;
std::atomic< int > length( 0 );
```

the queue itself, and a helper atomic counter (the queue itself does not provide an interface to obtain its length, as this cannot be done efficiently). The writer thread is implemented as follows:

```
struct Push : brick::shmem::Thread {
    void main() {
        int i = 0;
        while ( true )
            if ( length < L ) {
                q.push( i );
                ++ length;
                i = (i + 1) % (N - 1);
            }
    }
};
```

The main thread of the program then serves as the reader thread. We first initialise the queue and start the writer thread:

```
int main() {
    try {
        q = new Q;
        Push t;
        t.start();
```

The main reader loop checks that the items inserted by the writer come in the correct order. The implementations of `front` and `pop` perform additional assertions about the state of the queue.

```
        while (true)
            for ( int i = 0; i < N - 1; ++i ) {
                ASSERT_EQ( i, q.front( true ) );
                q.pop();
                -- length;
            }
```

Since the loop above is infinite, all we need to do is clean up in case the loop was never reached.

```
        } catch (...) {}
        delete q;
        return 0;
    }
```

note The try blocks in the program are somewhat peculiar, as is allocation of the queue on the heap. After all, we would normally construct the queue as a global variable, and at first sight there do not seem to be any exceptions involved. However, both these assumptions are false: memory allocation may always fail, and in C++ this leads to an exception. Moreover, the constructor of the queue allocates memory from the heap – if allocation fails in the constructor of a global object, there is no way to catch the resulting exception and the program will crash. Under usual circumstances, this is not a big problem – if there isn't enough memory to construct the global state of the program, it is unlikely that the program can do anything useful at all anyway. Nonetheless, DIVINE will notice this problem and report it. Since this is undesirable, we need to write our test code robustly.
While this may seem entirely superfluous, it might constitute a real problem in practical circumstances. If we consider programs that load shared libraries at runtime, presumably for non-critical functionality, the global constructors of the library will need to be executed at library load time. Since this might cause the program to crash even though it would be able to operate normally under given circumstances without the functionality provided by the library, this could be a serious problem.

We have compiled a few stats about the model used in verification of the IPC queue in the following table. Especially the time measurement is rather informal but should represent verification time on a typical development laptop. The results can be seen in **Table 3.1**.

B	L	N	states	walltime	memory
2	2	2	18 675	0:15	829MB
2	2	3	18 408	0:15	819MB
2	3	2	64 543	1:26	1 446MB
2	3	3	63 870	1:24	1 430MB
3	2	2	21 951	0:22	869MB
3	3	3	132 147	2:12	2 006MB

tab. 3.1 IPC queue verification.

note The implementation of the IPC queue dates back to 2008, and has seen extensive use ever since. It is heavily used by DIVINE in its default mode, and thanks to automated regression testing, there have been tens of thousands of successful runs of DIVINE on its regression test cases, and many other runs in different circumstances. Despite these 6 years of continuous use, our verification effort in 2014 has uncovered an error in the implementation.
The code treated the conditions `head == tail` and `head->next == 0` as equivalent, even though the implementation of `push` created a brief window where this was violated. The emptiness check used first of the two conditions but the code to shift the `head` pointer, quite naturally, checked the `next` pointer in `head`. In retrospect, this is clearly wrong, but despite heavy use and multiple reviews of the code, the problem only came to light as a counterexample in a verification attempt.

3.3 Termination Detection

Parallel algorithms with communication do not always posses simple locally-observable termination criteria, like their sequential counterparts. If termination criterion of a parallel algorithm is described through its global state, the individual constituent processes need to cooperate (and communicate) in order to decide whether the computation is finished. In many parallel algorithms, parallel searches being among those, there is a simple termination condition – there are no more items on the work lists: in graph exploration, this translates to "the open set is empty". Nevertheless, since the open set (or work lists in general) are distributed among the various participating processes, agreement must be achieved that no process has any work remaining to do. In distributed systems, this is made more complicated by the fact that parts of the open set may be represented by in-flight messages, already gone from process A, but not yet arrived in process B.

All in all, there are 3 different contexts in which we need to consider the problem of detecting termination: fully distributed memory, shared memory with message-based communication and fully shared memory. Each of these contexts requires different trade-offs.

In fully distributed memory, the solution is straightforward: use an established method, eg. Safra's algorithm [59]: whenever a process is active, the termination fails. When all processes are passive and message counts in the system are stable, the system is terminated if a second round of the distributed algorithm finishes successfully. In this case, it is the job of the message-passing infrastructure (MPI) to suspend passive processes in order to conserve resources.

With shared-memory parallelism, however, there is no such infrastructure in place, and our termination detection algorithm should be able to suspend execution of idle threads. For this reason, we use a relatively complex scheme, employing `pthread` condition-signalling primitives to allow suspension of idle threads until they receive work to do. The details of the scheme are laid out in **Algorithm 3.3**.

---

alg. 3.3 Like with the IPC queue, we show (skeleton) C++ code for the idleness check algorithm instead of pseudocode, in this case because we rely on pthread threading and locking primitives, which are hard to model in executable pseudocode. The entire algorithm runs with the global lock (mutex) held. Variables that start with an underscore are member variables of the class: their values are persistent across multiple calls into this code. The code runs in the context of any of the worker threads, whenever the thread decides that locally the criteria for termination are met.

```
MutexLock __l( _global_mutex ); // RAII
_done = false; // reset
bool done = true;
Set locked, busy; // a set of mutexes
```

The algorithm uses an array of mutexes, one for each worker thread. The first thing the algorithm does is to try locking these mutexes in a non-blocking manner (who points to the worker thread currently executing the algorithm):

```
for ( auto &i : _mutexes ) {
    if ( i.first == who )
        continue;
    if ( i.second->try_lock() )
        locked.insert( i.first );
    else
        done = false;
}
```

Next, if we managed to lock all the threads out using their respective mutexes, the algorithm checks whether any worker threads have work waiting (regardless of whether they are currently running or not):

```
if ( done ) {
    for ( auto &i : _mutexes )
    {
        if ( (i.first != who && i.first->isBusy()) ||
             i.first->workWaiting() )
        {
            busy.insert( i.first );
            done = false;
        }
    }
}
```

All the locks are then released, since we now have decided whether the system as a whole is idle or not.

```
for ( auto &i : locked )
    mutex( i ).unlock();
```

If the decision was that everything is done, the persistent _done variable is updated, so that all threads know that they can exit (no new work can appear in the system at this point) and they are all woken up.

```
if ( done ) {
    _done = true; // mark

    for ( auto &i : _conditions ) {
        if ( i.first == who )
            continue;
        i.second->notify_one();
    }
}
```

Otherwise, all threads that have work waiting (i.e. their IPC queues are not empty) are woken up and this thread (the one executing the check) is suspended until another thread decides to wake it up because work has accumulated while it was sleeping (or the system terminates).

```
if ( !done ) {
    for ( auto &i : _conditions ) {
        if ( !i.first->workWaiting() )
            continue;
        i.second->notify_one(); // wake up this thread
    }

    if ( !who->workWaiting() ) {
        __l.unlock();
        condition( who ).wait( mutex( who ) );
        __l.lock();
    }
}
```

The final result is stored in the `_done` variable: if the system terminated while this particular thread was sleeping (i.e. `_done` is true, regardless of the value of `done`), the thread can safely exit.

Finally, in the case of a fully shared-memory system, we have the benefit of dynamic load-balancing, hence it never happens that a subset of threads would be idle while another subset of threads would be busy. This removes the need to suspend idle threads, and makes the algorithm fairly simple: all it takes is to maintain a single counter representing the total amount of unfinished work in the system. Clearly, maintaining this counter naively would result in a sort of "ping-pong" between the participating CPU cores updating and testing the value all the time.
In order to mitigate this update problem, the counter is only approximated, adding and subtracting work in fairly big steps. The exact value of the counter is only computed when it is suspected that the computation may have terminated (eg. fetching items from

a shared work queue fails). To this effect, all threads locally maintain their approximation error – the amount by which they caused the shared counter to be over-estimated. When there appears to be no work left, all threads will in turn adjust the shared counter down towards its exact value – if it becomes 0, the system has terminated.

3.4 Shared Queues

Queues are a basic building block for parallel graph exploration algorithms, as those are usually based on some form of breadth-first search. Graph exploration in shared memory is most efficient when both the open and the closed set are shared by all threads. In this case, the vertices do not need to be statically partitioned and assigned to worker threads, but the work can be balanced dynamically – any work done by one thread is immediately and transparently seen by all other threads. In this kind of implementation, the queue is both a representation of the open set, and at the same time an instrument of load-balancing.
The first thing we notice is that all workers will be trying to push items onto one end of the queue at once, and likewise, all workers will be trying to pull items out from the other end – all at once. This means that whatever representation we choose for the queue, its endpoints will be subject to considerable contention. In practice, this means that inserting or removing individual elements is not a viable strategy, as the contention overhead would dominate and worker threads would experience so much latency on queue operations that we would be unlikely to be able to leverage any parallelism at all.

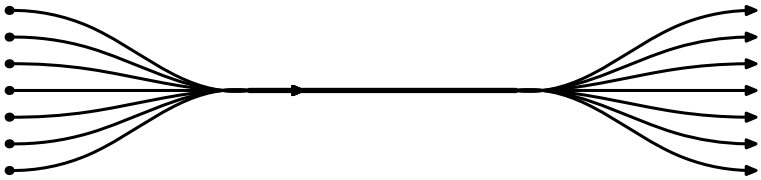


fig. 3.1 A multi-headed shared queue.

The natural design stemming from this consideration, then, looks like a rope frayed at both its ends: each thread has its own end for both reading and writing to the queue, and only when the length of this private end is sufficiently large (or small, on the reading end), it is twisted into (or twisted out of) the common "trunk" that is shared by all workers (cf. **Figure 3.1**). Clearly, this will only work for a sufficiently long queue: this corresponds to the intuitive understanding that in a very narrow graph, it is basically impossible to leverage any parallelism in the search.
While it is hard to imagine a significantly different basic design, there are still multiple options on how to actually implement the idea, and a few parameters to tweak for optimal performance. If the frayed ends are too short, we will see more contention – however, if they are long, we risk compromising load balance, as some threads fail to extract any items to process due to the entire queue being tied up in thread-specific areas.

The last question to answer is how to actually represent this frayed queue in terms of memory organisation. We can imagine the entire queue being a singly-linked list, with two "hubs" where the fraying happens. Whenever a strand becomes too long, it can be spliced into the shared section of the queue. Likewise, when a strand becomes too short, a section of the shared section can be atomically spliced into it. This clearly achieves the goal of reducing contention at the ends – splicing a section of a pre-built linked list is, afterall, a single atomic operation from the point of view of the shared section of the queue. There is, however, a downside to this approach: it has poor cache locality.

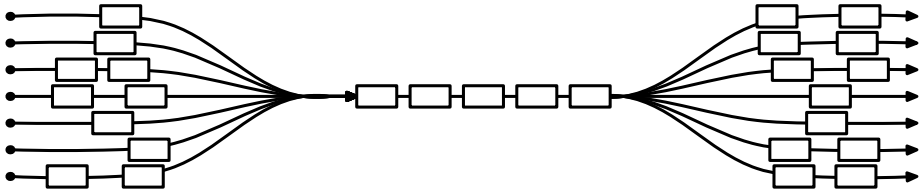


fig. 3.2 A chunked, multi-headed shared queue.

Like with *IPC* queues (**Section 3.2**), we would like to clump together at least a cache-line worth of items in a contiguous memory block, ideally more. While it is difficult to imagine a fully concurrent queue that is not at some level represented as a linked list, like it was the case with an IPC queue, nothing prevents us from chaining contiguous blocks of items into this linked list. This also happens to play well with the idea of per-thread strands – each such strand can be represented as a contiguous block, perhaps a vector of items. The shared section of the queue then becomes a linked list of just such blocks, appending and removing blocks – or chunks – as its units (see also **Figure 3.2**). Finally, depending on workload, it may be the case that the shared section of the queue is in fact accessed only very rarely. In such case, it may ostensibly make sense to drop the requirement for lock-free manipulation at the hubs, winning a modest improvement in memory efficiency (when a lock is involved, we can safely use a deque for the list of chunks instead of a linked list, which saves a few pointers – memory locality is unlikely to be a concern in this case, as most of the time, threads will alternate in accessing a particular hub).

3.4.1 Verification

Unlike the IPC queue, which is a lock-less design, the shared queue uses a lock-based approach: adequate performance is instead recovered by coalescing many similar operations (push and pop) into chunks which only need a single lock/unlock. Since all access to the queue is guarded by RAII-style locks, correctness is reasonably easy to ensure through code inspection. This does not preclude verification effort as such, but is not a high-priority undertaking either; as such, verification of the code has been relegated to future work.

3.4.2 Benchmarks

To asses the trade-offs involved in the design, we have created a few benchmarks, the results of which are shown in **Figure 3.3**. We have used four different implementations: *spinlock* is a simple spin-lock protected deque (the same that we used in comparison of IPC queue implementations), *lockless* is a lock-free implementation provided by Intel TBB [96], *hybrid* is a chunked implementation with a lock-free shared section of the queue while *chunked* is a chunked implementation with deque-based, lock-protected shared section.
Benchmark results for different object sizes are shown in **Section B.1**.

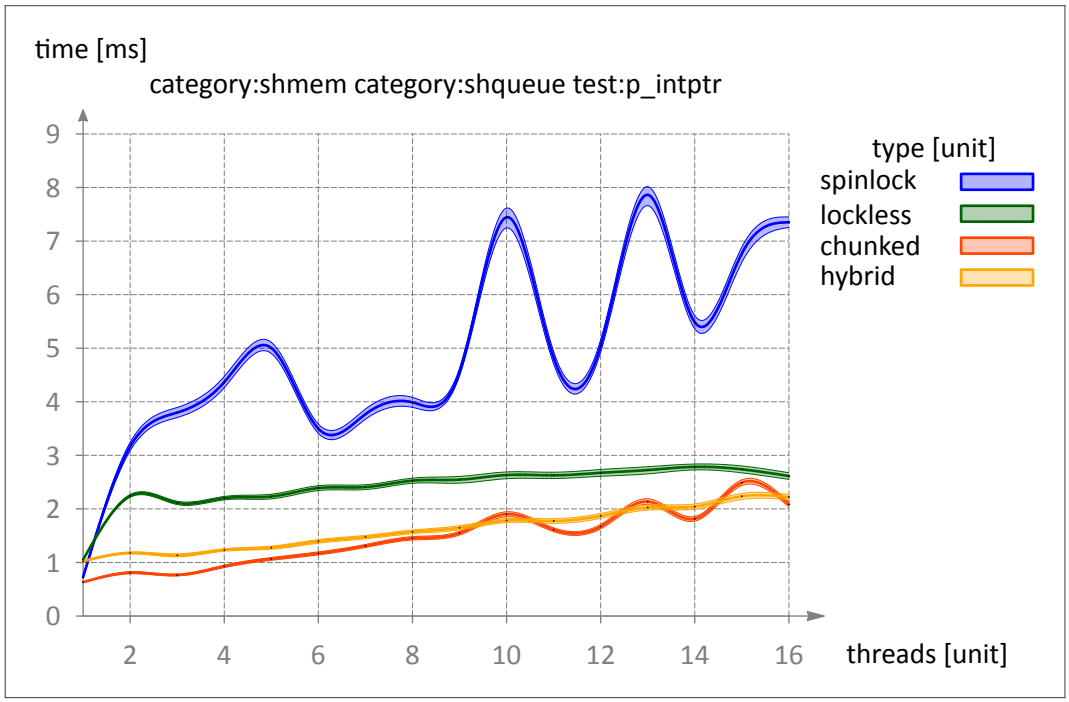


fig. 3.3 Scalability of various shared queue implementations.

3.5 Hash Tables

Hash tables comprise a very efficient representation of sets, and in model checking, they are the data structure of choice for representing the closed set during graph exploration. A hash table is represented as a vector of values in memory, associated with a function that maps *keys* to indices within this vector. The function is known as a *hash function* and should posses a number of specific properties: the distribution of key images should be uniform across the entire length of the vector, a small change in the key should produce a large change in the value, the function should be fast to compute and such a function should be available for arbitrary index size.

In practice, to implement the last criterion, hash functions for hash tables are usually implemented over the range of all 32 (64, 128 bit) integers in such a way that the remainder of division by an arbitrary integer n (or at least a power of two) will yield an uniform distribution in $\{1, ..., n\}$. The current practice is to use a purpose-built lookup function, either providing 64 (`lookup2` [97] or the more recent `lookup3` [98] are good candidates) or even 128 bits of output (the currently best available are spooky hash [99] and the city hash [130]).

3.5.1 Open vs Closed Hashing

Even with the best lookup function, hash collisions, and more importantly, index collisions will happen in a dynamic hash table. Hence, an important part of the hash table design is how to deal with such collisions, and there are two main options: open and closed hashing. With a closed hashing scheme, each position in the hash table is a "bucket" – capable of holding multiple values at the same time. This is implemented using an auxiliary data structure, usually a linked list. While closed-hashing is easier to implement and to predict, it usually gives poor performance. An alternative is to make each position in the table only hold at most one value at a time, using alternate positions for items that cause a collision. Instead of using a single fixed position for each value, the hash table has a list of indices. The most common such series are $h + ai + b$ where i is the sequence number of the index, h is the index assigned by a lookup function and a, b are arbitrary constants (a linear probing scheme). Another common choice is $h + ai^2 + bi + c$, obviously known as quadratic probing. An important property of a probing scheme is that it does not (significantly) disrupt the uniform distribution of values across indices. In case of a quadratic function and a hash table with a size that is a power of 2, a simple set of constraints can be shown to give a good distribution [23].

3.5.2 Cache Performance

There are many considerations when choosing a good hash table implementation for a particular application. In model checking, the hash table becomes very big, and as such, it cannot fit in the CPU cache entirely. For that reason, it is very important that all hash table operations have as much spatial and temporal locality as possible, to make best possible use of the CPU cache. The very nature of a hash table means that multiple insert or lookup operations on different keys will end up in entirely different memory regions: this is unavoidable. However, with a naive implementation, even a single lookup or insert can cause many cache misses: a closed-hashing scheme, for example, will need to traverse a linked list during collision resolution, which is a notoriously cache-inefficient operation. Even if we would use a different auxiliary data structure, we would still face at least one level of indirection, causing an extra cache miss. With open hashing and a linear probing function, we can expect a high degree of spatial locality in the collision resolution process: all candidate positions can be fetched in a

burst read from a continuous block of memory. In fact, this is a cache-optimal solution, as it only incurs the one unavoidable initial cache miss per lookup.
However, linear probing has other problems: the property that makes it cache efficient also means that it has a strong tendency to create uneven key distribution across the hash table. The clumping of values makes the collision chains long, and even though it is cache-efficient, the linear complexity of walking the chain will dominate after reaching a certain chain length. In contrast, a quadratic scheme will scatter the collision chain across the table. Hence, as a compromise, a hybrid probing function can be used: a quadratic function with a linear tail after each "jump": $h + q(i/b) + i\%b$ where q is a quadratic function. This has the advantage of scattering keys across the table, but in small clumps that load together into cache, without seriously compromising uniformity.

3.5.3 Variable-Length Keys

If there is substantial variation in key size, it is inefficient to store the entire key inline in the hash table, and impossible if no upper bound on key size is known. This means that we need to store pointers in the table and the key data becomes out-of-line. Unfortunately, this has disastrous effects on cache performance: each key comparison now requires an extra memory fetch: in order to find a key in the table, we need to compare it to each element in the collision chain.
To negate this effect, we can store the actual hash value of each key inline in the table: this way, we can first compare the hash values, without incurring a memory fetch. In vast majority of cases, a 64-bit hash will only test as equal if the actual keys are equal – we will only pay the price of an extra memory fetch in the cases where the keys are actually equal, which is at most once per lookup, and in only a tiny fraction of cases where the keys are distinct.
Even though efficient, this approach doubles the memory overhead of the hash table, storing a pointer and an equal-sized hash value for each key. This is especially problematic on 64-bit machines, making the overhead 16 bytes per slot when using a 64-bit hash value. Moreover, a 64-bit hash value is needlessly big, a much smaller, 32 or even 16 bit value would provide nearly the same value in terms of avoided cache misses. On most platforms, though, this will require arranging the hash table in terms of cache lines, as 96 or 80 bit slots will cause serious mis-alignment issues. With the knowledge of a cache-line size, we can organise the hash table into "super-slots" where each super-slot fits in a cache line, and packs the pointers first and the corresponding hash values next, in the tail.
This way, we can store 12 slots of 64-bit pointer + 16-bit hash values in a typical 128-byte cache line, or 10 slots of 64-bit pointer + 32-bit hash value. On architectures with 64-byte cache lines, this becomes 6 and 5, respectively. The downside of this approach is that it requires a rather peculiar hash table layout, or alternatively, a careful use of manual memory prefetching together with a two-vector representation (one vector for pointers, another for hash values).

On 64-bit machines, though, there is another option, which avoids most of the layout complexity at the table level. Contemporary CPUs only actually use 48 bits out of the 64 bit pointer for addressing, the rest being unused. While it is strongly discouraged to use these 16 extra bits for storing data (and CPU vendors implement schemes to make it hard), this discouragement is more relevant at the OS level. At the expense of forward portability of the hash table implementation, we could use these 16 bits to store the hash value, reconstructing the original pointer before dereferencing it. Finally, it is also possible to use an efficient pointer indirection scheme, which explicitly uses 48-bit addressing (see also **Section 3.7**) in a portable, forward-compatible fashion.

3.5.4 Capacity & Rehashing

As we have already said, a hash table is normally implemented as a vector, whether it contains single-value slots or multi-value buckets. As such, this vector has a certain size, and as keys are added into the table, it becomes increasingly full. The ratio of slots taken to slots available is known as a load factor, and most hash table implementations perform reasonably well until load of approximately 0.75 (although factors as high as 0.9 can be efficient [70]) is reached. At certain point, though, each hash table will suffer from overlong collision chains. This problem is more pronounced with open hashing schemes: in extremis, if there is only one free slot left, an open hashing scheme may need to iterate through the entire vector before finding it. There are two options on how to avoid this problem: the more efficient is to approximately know the number of keys that we'll store beforehand. However, this is often impossible, and in those cases, we need to be able to resize the table. This is usually done in the manner of a traditional dynamic array, only the values are not copied but rehashed into the newly allocated vector, which is usually twice the size of the current one.
Rehashing the entire table is at best a linear operation, but amortizes over insertions down to a constant per insert. In real-time applications, gradual rehashing schemes are used to avoid the latency of full rehashing. However, in most application, latency is of no concern and monolithic rehashing is in fact more efficient. As a small bonus, rehashing the table will break up existing collision chains and give the table an optimal uniform layout.

3.5.5 Concurrent Access

As we have discussed, open hashing is more cache efficient, and compared to a simple closed hashing scheme is also more space efficient. However, closed hashing has an important advantage: linked lists are a data structure easily adapted for lock-free concurrent access. Hence, most concurrent hash table implementations are based on closed hashing. The situation with open hashing is considerably more complex. It is relatively straightforward to implement a fixed-size hash table (i.e. for the scenario where we know the size of the working set in advance). In DIVINE, we have implemented a (nearly) lock-free, resizable open-hashed table [155], to retain the advantages of

open hashing, while at the same time being able to share the closed set among multiple threads.
Let's first discuss how a fixed-size open-hashed table can accommodate concurrent access. The primary data race in a non-concurrent table is between multiple inserts: it could happen that two insert operations pick the same free slot to use, and both could write their key into that slot – this way, the insert that wrote later went OK; however, the first insert apparently succeeds but the key is actually lost. To prevent this, write operations on each slot need to be serialised. The simple way to achieve this is with a lock: a spinlock over a single bit is simple and efficient on modern hardware, and since each hash table slot has its own lock, contention will be minimal. Using a lock is necessary in cases where the key cannot be written atomically, i.e. it is too long. If the key fits within a single atomic machine word, a locking bit is not required, and an atomic compare-and-swap can be used to implement writing a slot. When a lock is used, the lock is acquired first, then the value to insert and the locked slot are compared and possibly written. When using a compare-and-swap, in case it fails, we need to compare the keys – concurrent inserts of the same key could have occurred, and the same key must not be inserted at two different indices.
Concurrent lookups are by definition safe, however we need to investigate lookups concurrent with an insert: it is permissible that a lookup of an item that is being inserted at the same time fails, since there is no happens-before relationship between the two (this is in fact the definition of concurrency). It can be easily seen that an insert of a different key cannot disrupt a lookup of a key that is already present: all inserts happen at the end of a collision chain, never in the middle where they could affect a concurrent lookup.
In cases where variable-length keys are used based on the scheme presented in **Section 3.5.3**, lock-free access is only possible for variants where the pointer and the hash (if present) are located next to each other in memory, i.e. a hash-free (pointers only) table, or the 64 bit + 64 bit variant (only on machines with atomic 128-bit compare-and-swap), or the variant with pointer and hash combined into a single 64 bit value.

3.5.6 Concurrency vs Resizing

However, the scheme outlined in last section does not take the need for resizing and subsequent rehashing into account. The first problem of a concurrent resize operation is that we cannot suspend running inserts, as this would require a global lock. However, insert as a whole is not, and cannot be made, an atomic operation: only the individual probes are atomic. As a consequence, if we were to re-allocate the table at a different address and de-allocate the existing one, a concurrent insert could be still using the already freed memory. Since we cannot interrupt or cancel an insert running in a different thread, nor can we predict when will it finish, the best course of action is to defer the de-allocation. Unfortunately, even if we avoid writing into invalid memory, the same set of circumstances can cause an insert to be lost, since at the point it is written,

the copying (rehashing) of the table might have progressed beyond its slot (and since the probing order is not, and cannot be made, monotonic, this cannot be prevented). In order to clean up unused memory as soon as possible, and to solve the "lost insert" problem, we can, after each insert, verify that the currently active table is the same as the table that was active when the insert started. When they are the same, no extra work needs to be done, and the insert is successful: this case is the same as with a fixed-size table. If however the active table has changed, the insert has to be restarted with the new table. Additionally, we can use the opportunity to also clean up the old table if it is no longer used – if there are no further threads using the table. To reliably detect this condition, we need to associate an atomic reference counter with each pointer. Finally, if an insert has been restarted and succeeds, but the reference count on the old table pointer is not yet zero, the thread doing the insert can optionally help rehashing the table.

3.5.7 Verification

For verification of the concurrent hashset implementation, we have opted for a property parametrised with two numbers, T – the number of threads accessing the shared data structure, and N – the number of items each of those threads inserts into the data structure. The item sets inserted by each thread are disjoint.
First, we define a few types, namely the `hasher` which defines how to compute hashes from items and a few other item properties, like which value of the item type (`int` in this case) represents non-existence. The `HS` type represents the test hash table itself, and `t` is a pointer to the instance of the hash table we will be using for expressing the property.

```
using hasher = brick_test::hashset::test_hasher< int >;
using HS = brick::hashset::Concurrent< int, hasher >;

HS *t = nullptr;
```

With those definitions out of the way, we define what our model thread is like: namely, it takes two parameters – from and to, and inserts those items into the hash table. When all the items have been inserted, it verifies that the items it has inserted are indeed present.

```
struct Insert : brick::shmem::Thread {
    int from, to;
    HS::ThreadData td;

    void main() {
        try {
            int i = from;
            for ( int i = from; i < to; ++i )
                t->withTD( td ).insert( i );
            for ( int i = from; i < to; ++i )
                assert( t->withTD( td ).count( i ) );
        } catch (...) {}
    }

    Insert() : from( 0 ), to( 0 ) {}
};
```

Finally, we need to set up the worker threads and the hash table. We use a table which is limited to 4 size increases (the hash table normally uses a factor 16 resize until it reaches the size of at least 512k cells, but this fast initial growth is suppressed for verification runs, using the normal 2-fold increase in each growth step). The initial size of the hash table is set to 2 cells, so that in most cases at least 1 resize is required during the test.

```
int main() {
    try {
        Insert th[T];
        t = new HS( hasher(), 4 );
        t->setSize(2);
```

We then proceed to set up parameters of the individual threads, setting their `from`/`to` parameters to non-overlapping, consecutive ranges.

```
        for ( int i = 0; i < T; ++i ) {
            th[i].from = 1 + N * i;
            th[i].to = 1 + N * (i + 1);
        }
```

The threads are then started, with the main thread taking a role of a worker thread as well, in order to reduce state space size by not spawning an extra thread when it is not needed. Since memory allocation can happen during thread creation, we need to guard the loop for exceptions and clean up in case of an allocation failure.

```
        int i;

        try {
            for ( i = 0; i < T - 1; ++i )
                th[i].start();
        } catch(...) {
            for ( int j = 0; j < i; ++ j )
                th[i].join();
        }

        try {
            th[T - 1].main();
        } catch (...) {}
```

When the main thread finishes its portion of the work, it waits for the other threads and cleans up.

```
        for ( int i = 0; i < T - 1; ++i )
            th[i].join();

    } catch (...) {}

    delete t;
    return 0;
}
```

In this scenario, we can observe the huge impact of the exponential state space increase. For $T = 3, N = 1$, verification of the above test-case took multiple days using 32 cores, generated over 716 million states and used about 80GiB of RAM. On the other hand, verification for $T = 2, N = 1$ finishes in less than 3 minutes, uses 1.4GiB of RAM and generates fewer than 100 000 states.
This means that out of the desirable properties, we were able to verify that a cascade of two growths (possibly interleaved) is well-behaved when two threads access the table – using $T = 2, N = 4$ – in this scenario, a single thread can trigger a cascade of 2 growths, while other threads are inserting items. We were also able to verify that a single growth is correct (it does not lose items) in presence of 3 threads. A scenario with cascaded growths and 3 threads, however, seems to be out of our reach at this time. Nevertheless, the verification effort has given us precious insight on the behaviour of our concurrent hash table implementation.

note While the hash table described in this section was in a design and prototyping phase we have encountered a race condition in the (prototype) implementation. The fact that there is a race condition was discovered via testing, since it happened relatively often. The problem was finding the root cause, since the observable effect of the race condition happened later, and traditional debugging tools do not offer adequate tools to re-trace

the execution back in time.[23] In the end, we used DIVINE to obtain a counterexample trace, in which we were able to identify the erroneous code.

3.5.8 Benchmarks

To assess performance of the final design with concurrent resizing, we have created a number of synthetic benchmarks. As the baseline for benchmarking, we used implementation of `std::unordered_set` provided by `libc++` (labelled "std" in results). Additionally, we have implemented a sequential open-hashed table based on the same principles as the final design, but with no concurrency provisions (tables "scs" and "sfs") – this allowed us to measure the sequential overhead of safeguarding concurrent access. Moreover, DIVINE uses this sequential-access-only implementation for cases where concurrent access cannot happen.

Since `std::unordered_set` is only suitable for sequential access, as a baseline for measuring scalability, we have used a standard closed-hashing table (labelled as "cus", from `concurrent_unsorted_set`), an implementation of a concurrent hash table provided in Intel Thread Building Blocks [96]. The final designs presented here are labelled "ccs" and "cfs". The middle letter indicates the size of the hash table cell c for "compact" and f for "fast": the "fast" variant uses a hash cell twice as wide as a pointer, storing a full-sized (64b) hash inside the cell. The "compact" variant uses a truncated hash that fits in the spare bits inside a 64-bit pointer. (The hash inside cells is only useful in hash tables with out-of-line keys; for integer-keyed tables, they are simply overhead).

As the common performance measure, we have chosen average time for a single operation (an insert or a lookup). For benchmarking lookup at any given load factor, we have used a constant table with no intervening inserts. Five types of lookup benchmarks were done: miss (the key was never present in the table), hit (the key was always present) and a mixture of both ($^1/_2$ hit chance, and $^1/_4$ hit chance). For insertions, we have varied the amount of duplicate keys: none, 25 %, 50 % and 75 %.

All of the insertion benchmarks have been done in variant with a pre-sized table and with a small initial table that grew automatically as needed. Finally, all of the benchmarks outlined so far have been repeated with multiple threads performing the benchmark using a single shared table, splitting workload equivalent to the sequential benchmarks, distributed uniformly across all threads. All the benchmarks have been done on multiple different computers, with different number of CPU cores and different CPU models, although we only report results from a single computer – a 12-core (2 sockets with 6 cores each) Intel Xeon machine.[24]

As a representative plot, we have chosen insertion with 50 % key duplicity rate, with 1 million items and a pre-sized hashtable with half a million cells, in **Figure 3.4**. All the remaining plots were relegated to **Appendix B**, **Section B.3**.

[23] An extension to `gdb` to record execution exists, but we were unable to use it successfully. Either the window in which time reversal was possible was too narrow, or the memory and time requirements too high.

[24] The full data set will be eventually published online, but is too extensive to fit even in an appendix of this thesis.

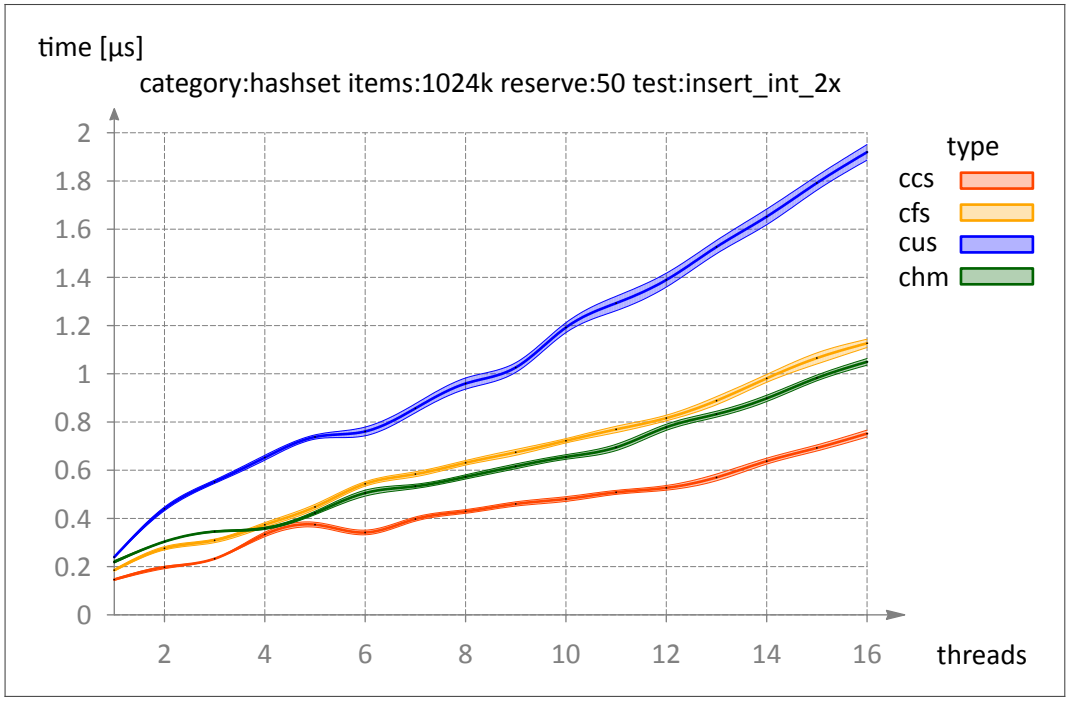


fig. 3.4 Scalability of insertion into various hash table implementations; each item was inserted twice on average.

3.6 Compression

Depending on the verification task, the storage size of a single vertex (state) can be fairly large. This is especially true of more complicated model checking inputs, like LLVM[25]. In those cases, as we have mentioned in **Section 2.5.2**, it makes sense to consider compression schemes for states and/or the entire state space. In DIVINE, we have implemented the latter [153], using a scheme similar to *collapse* [86]. Since our hash table is resizeable to facilitate better resource use, we cannot directly use some of the improvements that rely on fixed-size hash tables [110]. On the other hand, since the hash table we use can accommodate variable-size keys, we are not limited to fixed-layout trees and can use content-aware state decomposition like in the original *collapse* approach (but unlike original collapse, we can decompose the state recursively, which is useful with more complex state vectors, like those arising from LLVM inputs). The decomposition tree structure is illustrated in **Figure 3.5**.

Our approach uses three hash tables that are adaptively resized as needed. One holds root elements – one root element corresponds to each visited state 1:1. These root

[25] In theory, nothing about LLVM per se causes states to be large; in practice, however, inputs that are expressed in terms of LLVM have a tendency to have much richer state than more traditional formalisms, like DVE or ProMeLa.

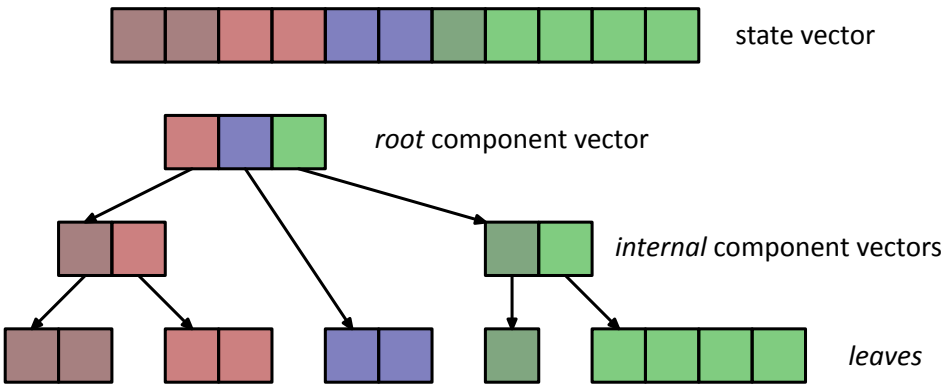


fig. 3.5 A decomposition of a state into a component tree. The leaves represent fragments of the original state vector.

elements are represented as component vectors, where each component is represented as a separate object in memory. Those components are de-duplicated using a *leaf table* – a state fragment that is identical in multiple different states is only stored once, and the *root* table refers to the de-duplicated instances of those objects. To facilitate recursive decomposition, we also maintain a third table, *internal*, for internal nodes of the state decomposition tree. The *internal* nodes have the same structure as *root* nodes (a vector of pointers), but they do not correspond to complete states and the *internal* table is not consulted by the model checking algorithm when looking up vertices during search.
The component vectors contain a flag to decide whether a particular component is another component vector or a state fragment, as otherwise they are not distinguishable – both are stored as raw byte arrays in memory, without distinct headers. Clearly, reconstructing a state vector from a component vector is easily done by walking the decomposition tree and copying leaf node content to a buffer from left to right. In theory, storing the size of the entire state in the *root* component vector could improve efficiency by making the reconstruction work in a single pass, copying fragments into a pre-allocated buffer. In practice however, the decomposition trees are small and the requisite pointers are retained in fast CPU cache on the first pass (when the buffer size is computed), making the savings from a single-pass algorithm small. Moreover, the extra memory overhead of storing another integer along with each state is far from negligible.

---

ex. 3.1 The trade-off inherent in tree-based compression schemes is visible in **Figure 3.5** and **3.6**. Compare the number of squares (memory cells) in these two pictures. The original state vector occupies 11 cells, its decomposition uses 18 cells. However, adding another similar state (state B in **Figure 3.6**) increases the memory use only by 9 cells in the compressed variant, while it would add another 11 cells without compression. The state vectors illustrated here are extremely small; real-world LLVM states typically occupy hundreds or thousands of memory cells and bigger states naturally favour

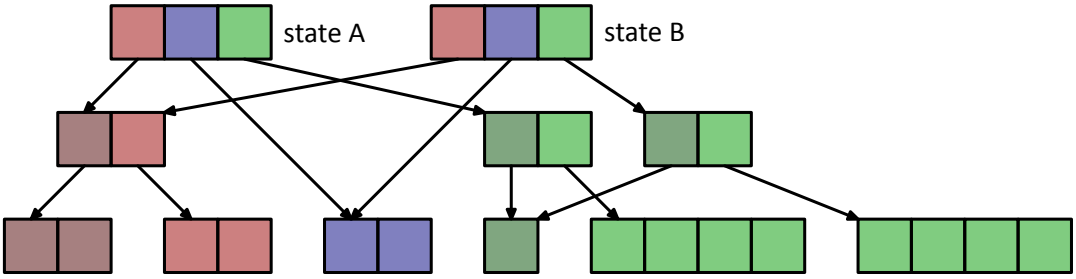


fig. 3.6 A de-duplicated pair of states. The layers are analogous to **Figure 3.5**. States A and B differ only in the light green component.

compression. On the other hand, a realistic implementation introduces slightly more memory overhead than the idealised picture show here.

3.7 Memory Allocation

Memory allocation is an extremely frequent operation in an explicit-state model checker. Moreover, the memory pool that threads allocate from is a shared resource, requiring certain amount of synchronisation. One way to side-step this issue is to statically pre-allocate as many resources as possible – this is the approach taken by, most prominently, the model checker SPIN. The main downside of this approach is that the tool either has to "guess" resource use very well ahead of time, or rely on the user to provide guidance. In all but very simple scenarios, the former is very hard to get right – models vary wildly from one to another in which parts of the model checker they stress. Some require very long queues or deep stacks, even when the overall size of the state space is comparatively small. Others only need a very small queue but the state space is huge, and almost all memory needs to be allocated towards the closed set. Some models have few big states, requiring few slots in the hash tables, but need a lot of memory for storing the states themselves.[26]

However, there is a more important limitation, namely with regard to multitasking: users expect to be able to execute multiple instances of a program at the same time, especially if the verification runs are well below the limits of the computer they are using. Static resource allocation in such cases becomes a chore – especially so if multiple users are involved on shared hardware. In most cases, we aim at interactive use: batch scheduling is only suitable for very large instances, where the entire computer (or a cluster) is tied up in a single verification task. Meanwhile, a large SMP system can easily serve many tasks and many users interactively – but this means that tasks should only consume resources that they actually need, so that resource conflicts are minimised.

[26] The LTSmin model checker avoids this particular resource split by storing state vectors decomposed, each fixed-size chunk stored inline in the large pre-allocated hash table.

This is very hard to achieve if memory needs to be pre-allocated at a time when the size of the state space is not yet known.
To address those issues, DIVINE uses dynamic allocation for all resources, achieving optimal hardware utilisation when multitasking. There are, however, multiple challenges associated with this flexibility, especially when dealing with parallel algorithms.[27]

3.7.1 Allocation Profile

When designing a custom memory allocator, the first thing to ask is what is the allocation profile of our target application. Are object sizes similar, or distributed across a wide spectrum? Are there many small allocations, or few big allocations? Is memory retained for a long time, or a short time? Is memory deallocated often?
We can answer most of those questions for DIVINE: for one, there is a tendency to see many objects of similar size. This is most visible in models with fixed-size states (this is actually the case with majority of input languages in DIVINE: most traditional modelling languages require all state variables to be explicitly declared and do not provide dynamic variables). It is also true, to a smaller extent, with variable-size state vectors: many states will differ in content but not the size of the state vector. For LLVM, state size changes when a thread is created, a function is entered or left and when a new thread is created. All these operations are comparatively rare, so we can expect many states of any given size to appear over time. This is even more pronounced when compression enters the picture, since the fragments have more uniform sizes than the entire state vectors. This favours a design where objects of a particular size are grouped into bigger blocks, causing fewer calls to the parent allocator.[28]

note This layout also offers the opportunity to store object size as allocator metadata, once per block of objects. When state vectors are of variable length, their length needs to be stored somewhere: if each state vector stores its own length, this either adds 4 bytes of overhead per state, or causes the rest of the vector to be stored unaligned. Both are far from optimal. If the size is stored once per block, a single 4-byte word can be used to keep the size for hundreds of objects, saving considerable amounts of memory. This scheme however means that the allocator needs to be able to find block metadata from a pointer, to read the object size associated with the pointer.

Second, there are two main classes of objects during state space exploration: the first class contains state vectors that are part of the closed set, and will be reclaimed at the end of the verification run, but not earlier. The second class contains newly generated successor states that may or may not be duplicates of states in the closed set – some of

[27] Intra-process parallelism can be very useful even when multiple verification instances are involved. A 64-core system can easily accommodate 4 verification tasks running on 16 cores each, splitting memory between those 4 tasks as needed. If memory becomes scarce, some of the processes can be suspended and swapped out to disk and later, when other tasks have finished, resumed again.

[28] In case of DIVINE, the parent allocator is the Intel's TBB [96] malloc, which in turn calls glibc's malloc to obtain memory in yet larger chunks.

those will go on to be added to the closed set (which may require their re-allocation if compression is enabled) while others will be deallocated when they are found to be duplicates. In other words, some objects are short-lived, and some are very long lived – however, there are few, if any, "in-between" objects. This split would favour a generational allocator – especially since we often know ahead of time whether a particular object will be short- or long-lived (at least in the case where compression comes into play – in other circumstances, the distinction is less clearly cut). Nevertheless, we eventually decided against using generational approach: the advantages of multiple generations are more pronounced in a garbage-collected scheme (whereas DIVINE releases memory explicitly), and there is still the question of extra overhead of moving objects across generations when compression is not in use. The possible advantage even in an explicitly managed heap would be the ability to pack the mature space more tightly and optimise the nursery for high allocation performance and best possible alignment.
Finally, the number of memory allocations is very high, so throughput of the allocator is very important. Latency, on the other hand, is not very interesting, as DIVINE is not in any sense a real-time application. This means that when there are opportunities to pre-compute particular operations for many objects at once, this should be preferred if it improves throughput at the expense of latency.

3.7.2 Pointer Representation

There are two basic options on how to represent pointers: either use raw machine pointers, or use an indirection scheme. The former has a clear advantage in terms of access speed: dereferencing a raw machine pointer is as fast as it gets – any other representation will incur additional costs. On the other hand, most contemporary platforms use pointers that are 64 bits wide – for realistic memory sizes, this constitutes substantial overhead. Current CPUs can physically address at most 48-bit memory addresses, while the rest of the pointer representation is unused – that is 16 bits of memory lost for every pointer. Moreover, there are plenty of places in DIVINE where extra bits packed inside pointers can save considerable amount of memory: the hash tables, for example, can use (some of) those 16 bits to store a small part of the hash value to avoid full object comparisons and speed up lookups considerably (see **Section 3.5.3**), at no extra memory expense. The compression algorithm can use a few bits for type-tagging pointers (see **Section 3.6**), making it free, in terms of memory use, to distinguish state vector fragments from state component vectors.
Moreover, a custom pointer representation enables the allocator to easily find the block header for any given pointer, making it possible to obtain object sizes from pointers to those objects. As explained in previous section, this can save considerable memory in some cases.
The main downside is that the pointer dereference operation needs to consult a lookup table to reconstruct the raw machine pointer. The lookup tables can be represented in such a way that this can be implemented using a single addition instruction, followed

by a memory fetch from the lookup table, followed by another addition instruction. Since the lookup tables are small, we can hope that they will always be readily available from fast CPU cache. In our informal testing, the slowdown from this indirection was in single-digits percent range, while the memory savings were considerable. Based on this, we have decided to use indirect pointers for storing states and state fragments.

3.7.3 Concurrent Access

Clearly, multiple threads need to allocate memory. The canonic way to ensure scalable multi-threaded allocation is to keep per-thread allocation structures and only synchronise threads when large blocks are requested from the parent allocator. Somewhat more importantly though, multiple threads need to *free* memory, without synchronising too often. This is more problematic, because one thread may allocate memory and another thread may need to free it. Normally, memory allocators use a data structure called a *freelist* to maintain the map of released but not yet re-used memory. Since our allocator is tuned for managing many objects of a limited selection of sizes, it is possible to maintain a separate freelist for each object size – this is by far the most efficient option. General-purpose allocators often need to use more complicated freelist representations in order to support more advanced heap maintenance in presence of highly variable object sizes.

note Freelists are often implemented *inline* – when a heap object is freed, the memory it occupied is rewritten with a freelist cell – in our case, the freelist is a singly-linked list with no additional structure. The downside of this approach is that each memory object needs to be at least as big as a pointer. For very small objects, this could mean significant overhead. However, since this is a special-purpose allocator, we know that DIVINE rarely needs to allocate objects significantly smaller than a single pointer – even very simple models usually have states at least 8 bytes long.

As mentioned above, a concurrent allocator needs to be able to efficiently share the freelists among multiple threads. There are two basic freelist operations: *insert* and *remove*. First is used when memory is freed and the reclaimed memory needs to be tracked so it can be re-used later. The latter is used when an allocation request is made – if there is a non-empty freelist for the right object size, a cell is removed from that freelist and used to satisfy the request. New memory is only allocated when the corresponding freelist is empty. When the freelist is represented as a singly linked list, it can be used either as a queue or as a stack: in case the *insert* and *remove* operations are performed on opposite ends of the list, the freelist behaves as a queue and when they happen on the same end, it behaves as a stack. In most cases, stack semantics are the more suitable choice: they are easier to implement (there is no need to maintain a separate pointer to the end of the list) and newly re-used block have usually been freed recently; as such they stand a better chance of being still present in CPU cache.
At first sight, a queue-style freelist may offer better access from multiple threads – after all, it provides two distinct locations for the *remove* and *insert* operations, reducing the

amount of synchronisation that is required if multiple threads access the same freelist at once. However, this is far from enough: in a multi-threaded environment, there will be many *insert-insert* and *remove-remove* conflicts. In all but very special circumstances, the freelists cannot be entirely shared. The correct approach is very similar to what we have used in the implementation of a shared queue (see **Section 3.4**). We set a threshold n (the exact value is subject to fine-tuning), which dictates how big a per-thread freelist can become – after reaching this size, the freelist is abandoned by its owner thread and moved to a shared list of freelists, and a new empty freelist is created for local use.
This means that we have 2 types of freelists in the system: private, per-thread freelists – one for each object size in each thread. Each private freelist contains at most n objects. Additionally, there are *shared* freelists – these are always of the same size, each containing *exactly* n objects. When a thread needs to allocate memory and its private freelist for the particular size is empty, before resorting to obtaining fresh memory, it checks whether a suitable shared freelist exists. If this is the case, it claims ownership of that freelist, changing its status to *private* and satisfies the allocation request from this newly private freelist.
This way, *insert* and *remove* operations of a single thread always share the same location – maximising cache re-use, but multiple threads never use the same location for any freelist operations, minimising synchronisation overhead.

3.7.4 Implementation

The considerations laid out in previous sections give us a fairly good guidance on how to implement an efficient allocator for use in DIVINE. Our implementation uses a custom pointer type, which is translated to machine pointers on demand, at the cost of an extra memory fetch (which is expected to be served from cache, since the indirection table is usually very hot) and a couple of addition instructions. All data structures in the hot paths of the allocator (object allocation and deallocation) are thread-local and expensive thread synchronisation only happens in special circumstances, usually after some threshold is exceeded: either per-thread freelists have grown too big, or they have become empty; or when all freelists are empty and no pre-allocated memory is available, in which case it needs to be obtained from the parent allocator.
New memory is obtained in blocks of adaptive size. Even though we *expect* that each object size will contain many different objects, this is not always, strictly the case: one particular corner-case is LLVM models with compression. Even though many states of the same size are allocated and later deallocated, their lifetimes only rarely overlap – therefore, it is advantageous to first allocate blocks for only a few objects of a given size. The overhead per object is higher in those cases, but if many objects of the same size are allocated later, the extra cost of the first few blocks is quickly amortised by allocating larger and larger blocks over time.

The shared data structures: indirection tables and lists of shared freelist, are implemented as standard lock-free data structures. Since they are only accessed comparatively rarely, no special precautions need to be taken to make access to them more efficient – the indirection table is almost entirely read-only – it is only written when a new block is allocated. Additionally, a shared counter is maintained to assign blocks to threads (threads claim 16 blocks at once to minimise contention on this counter; the blocks are only allocated when they are needed though).

4 LLVM

In this chapter, we will focus on LLVM bitcode, its role in the LLVM program compilation framework, its semantics, and our adaptation of DIVINE to work with it as an input language.

4.1 Language & Bitcode

LLVM is, foremost, a toolkit for building compilers. It consists of a set of libraries for building an intermediate representation of a program, a large number of transformations on this intermediate representation, and a number of platform-specific code generators, which output a CPU-specific assembly program.
This is very similar to how all modern compilers work. The interesting difference that we can exploit for model checking is that the intermediate language has a specification and a stable external representation. The libraries can load this external representation, called LLVM bitcode, and work with it the same way as a compiler middle-end would work with the output of a front-end. Finally, LLVM libraries provide a simple interpreter for this intermediate representation, which is very unusual in compilers.
The language is, in a way, similar to a traditional assembly language of many CPUs, but also has some crucial differences. An LLVM bitcode file contains global data specification and a number of function definitions. Each function is represented as a control-flow graph, in terms of basic blocks.

def. 4.1 A basic block is a sequence of machine (or LLVM) instructions with no branching. The computation of a program within a basic block is entirely sequential, performing instructions in their order of appearance in the block. The final instruction of a basic block may be a branching (jumping) instruction. ■

The instructions in each basic block operate on values in virtual registers, or they can move values from registers to memory or vice-versa. Each instruction has at most one output value which is stored in a virtual register upon execution of the function, and any number of inputs which must all reside in registers, or can be provided in the form of constants. Memory access is implemented through `load` and `store` instructions, which take an address stored in a register.
In addition to memory access, the instruction set of LLVM includes the standard fixed--point and floating-point arithmetic operations, bitwise operators, control flow (branching, function calls and exception handling), comparison operators and value conversions.

---

ex. 4.1 Let's have a look at a fragment of an LLVM bitcode program. Here, `@i` is name of a global value (containing a memory address), `%n` are names of registers, `br` is a branching

instruction. `i32` is the data type – a 32bit integer – stored in registers `%5` and `%10`, `i32*` is the type – a memory address of a 32 bit integer – of the value stored in `@i`.
Bitcode is naturally broken up into basic blocks (see **Definition 4.1**):

```
<label>:4
  %5 = load i32* @i
  %6 = icmp slt i32 %5, 2
  br i1 %6, label %7, label %4
```

Different basic blocks are connected via jump instructions, in this case `br` in a three-parameter form, i.e. a conditional jump. It might either loop back on its own start (label 4) or continue on to the second basic block:

```
<label>:7
  %9 = load i32* @i
  %10 = add nsw i32 %9, 1
  store i32 %10, i32* @i
  br label %4
```

4.1.1 Control Flow Instructions

In LLVM, like in most machine languages, there are a few instructions implementing control flow, namely conditional and unconditional branching (or jumping; both direct and indirect), call, return and invoke (a special variant of call for use with exception handling). All control flow instructions other than call/invoke are terminator instructions: they always come last in a basic block. Moreover, the target of a jump in LLVM is a basic block: it is not allowed to jump into the middle of a basic block.

4.1.2 Single Static Assignment

LLVM bitcode is always in a (partial) SSA form: each register is only assigned (defined) once. However, this is only true of register values: address-taken variables (i.e. those that can become targets of a pointer) are not part of the SSA and exist in a separate space as normal "mutable" variables. In straightforward, unoptimised LLVM bitcode, each C-level variable is an address-taken variable, created by an `alloca` instruction. Only compiler-created intermediate values become register values. However, the LLVM optimiser can (and will) lift many address-taken variables into registers if the addresses of those variables are not *actually* taken.
In order to allow non-trivial control flow, SSA needs so-called φ instructions, or nodes. These instructions may appear in the head of a basic block whenever multiple distinct branching instructions contain this basic block as a target. The semantics of the φ node

is then to create a new register, with content copied over from another pre-existing register *depending on which other basic block transferred control to this basic block*.

4.1.3 Data and Registers

Both values and variables are *typed* in LLVM: the types are either primitive/scalar (integer values with specific bit widths, floating-point numbers, pointers) or aggregate: arrays, vectors and tuples. Many instructions put restrictions on data types that their operands can take, eg. integer arithmetic requires integral arguments, and provides an integral result. Additionally, LLVM provides explicit type casting, bit-extension and bit-truncating operations. Unlike first-class values in registers, *memory* is untyped in LLVM. Storing a value at a particular address and loading the same address into a register of different type is permissible, and is equivalent to a `bitcast` operation. This somewhat reduces the strength of the type system, as it only applies to the SSA portions of the IR. Since layout of aggregate types is a complex issue – depending on particulars of the target architecture and the target ABI[29] – LLVM provides primitive operations to compute offsets into aggregate types, `getelementptr` and to extract/insert individual components out of aggregate values stored in registers (`insertvalue` and `extractvalue`). Unfortunately, clang does not exclusively use `getelementptr`, and as such, we need to make sure that the LLVM bitcode layout matches clang's expectation. For this reason, we have to respect the layout of structures used by LLVM– even though it would be possible for DIVINE to pack certain values more tightly than they are in usual circumstances. This most importantly includes 64-bit wide pointers in LLVM bitcode generated in 64-bit mode of clang.

4.1.4 Exception Handling

Exception handling is an area where the code generator needs to cooperate in order to implement correct language semantics. Since code generators are part of LLVM, but LLVM itself is programming-language-agnostic, the LLVM code generators need to provide a sufficiently generic interface to allow implementation of efficient exception handling.
In all modern C++ compilers, zero-cost exceptions are the norm: the exception handling machinery imposes no overhead at all unless an exception is actually thrown. This means that the code generator is not allowed to insert special instructions for calls or for saving context when entering try blocks. In order to allow this sort of behaviour, all exception handling logic needs to happen at an exception throw time, and for this to be possible, a stack unwinder is required. The unwinder is platform-specific, and needs to understand the particular ABI and most importantly the layout of the program stack

[29] Application Binary Interface, in this context meaning the set of rules in use for data layout, calling conventions, endianness, etc. Some CPU architectures allow multiple incompatible ABIs on the same chip, most prominently the ARM family of CPUs.

and individual stack frames. LLVM itself does not provide an unwinder library: it is usually provided by the operating system.
Unfortunately, the interface of the unwinder library is not entirely specified, and as such, it is also somewhat platform-specific. There are two major surfaces of the unwinder, each exposed to different part of the compiler/standard library duo. On one hand, the unwinder needs *unwind tables* in order to correctly unwind the stack. These unwind tables are generated by LLVM, since they reflect the high-level structure of individual stack frames, which is itself generated by LLVM. These tables end up being a part of the *program text*, i.e. they are stored in the executable image, and are as such a static part of the program. On the other hand, there is the "dynamic", or runtime, interface of the unwinder library, which is exposed to the language runtime instead: when an exception is raised, the language runtime uses the unwind library and the unwind tables generated by LLVM to guide the exception handling process.
While C++ is the primary target of the exception-handling mechanisms in LLVM, care has been taken to make it sufficiently general to accommodate other language runtimes, as long as their exception handling works along the same general principles. The main requirement for an exception system to be compatible with LLVM is that it can use the same unwinder interface, or at very least that it can process the unwind tables produced by LLVM. On many platforms (all modern UNIX systems based on the ELF executable format), these unwind tables are in a standardised format, mandated by the DWARF specification [53]. Other platforms use different unwind tables, though.
Besides information about the structure of a stack frame, unwind tables contain information about how exception handling should process this particular stack frame. In programming languages with lexical scoping, lexically scoped variables cease to exist when their scope terminates: normally, this happens when a function returns. However, exceptions create a new way in which a lexical scope can cease to exist, namely that an exception is propagated through this scope upwards. As long as lexically scoped (local) variables are sufficiently simple (plain old data in C++ terminology), this is not a major problem: the stack is unwound, so the storage associated with those variables is automatically reclaimed. However, C++ and a number of other languages allows scoped variables of complex types, with associated destructors: code that the runtime guarantees is executed just before the variable is deallocated. Particularly in C++, this is widely used to implement reliable, automatic resource acquisition and release[30]. Even though similar schemes have been proposed for C [148], they are usually implemented using `setjmp` and `longjmp` primitives, do not use any compiler support and therefore do not map to the LLVM exception handling mechanism.

---

ex. 4.2 A C++ program which works with some sort of a managed resource (in most programs, at least heap memory works in this way) may use a definition along these lines:

[30] In the C++ community, this design pattern is known as RAII: Resource Acquisition Is Instantiation. Among other things, it is used to safely hold mutual exclusion locks, dynamically allocated memory and other non-composable resources inside functions that could experience non-local loss of control due to exceptions.

```
struct R1 { R1() { /* ... */ } ~R1() { /* ... */ } };
struct RC { R1 r1; int *resource;
  RC() : r1() {
    resource = new int[32];
  }
  ~RC() { delete[] resource; }
};
```

The destructor of the class frees up a locally acquired resource. In fact, the standard C++ library provides templates for exactly this purpose (`std::auto_ptr` and in C++11 `std::unique_ptr` and `std::shared_ptr` which also provides reference counting). Since the C++ runtime guarantees invocation of destructors of local objects when their stack frame is destroyed, the resource is safely freed even if the owner of the object does not take any special precautions against exceptional situations. The user code could look something like this:

```
int main() {
  try {
    RC res;
    // work with the resource...
  } catch ( ... ) {
    // handle exceptions
  }
  // some other code here; res is no longer in scope
}
```

When an exception is thrown, or when control normally leaves the scope of the try block, the dynamic memory is correctly freed by the destructor of `RC`. Moreover, in the case where the resource allocation in the constructor of `RC` fails, but the constructor in `R1` has already allocated its resource, the `R1` destructor is invoked on the partially constructed object.

Nevertheless, LLVM as such has no concept of destructors, nor does the unwinder library. The language compiler needs to generate *cleanup handlers*, i.e. blocks of code that take care of calling any appropriate destructors, or performing other language-specific cleanup when a stack frame is torn down because the stack is being unwound. Moreover, the same mechanism is used for *exception handlers*: the main difference is that an exception handler *stops* the propagation of an exception, and its role is to deal with the exceptional situation: exception handlers correspond to the *catch* blocks attached to a *try* block.
In order to improve efficiency (at the expense of simplicity) of the unwinder, it has a concept of *exception type*: different types of exceptions can happen, and a particular catch block may handle only a subset of those exception types. Each call-site in

each call frame possibly contains a cleanup handler, and a list of exception handlers. Deciding whether a particular exception handler can handle a particular exception type is deferred to a *personality function*: a language-specific callback provided to the unwinder. This personality function helps the unwinder decide, among other things, which handler to invoke for a particular exception type.

4.1.5 Mapping Exceptions to LLVM

Now that we have established the basics of how exceptions are implemented in general, we will look at how those concepts map to LLVM. The machinery provided by LLVM to handle exceptions consists of 3 instructions: `invoke`, `landingpad` and `resume`. The `invoke` instruction is like a `call` instruction, but it provides extra provisions for exception propagation: unlike `call`, it is a terminator instruction, i.e. it is always last in a basic block. It is also a branching instruction: it takes two basic block addresses as parameters corresponding to two branches – the first is taken upon a normal return from the function, the other is taken if an exception has been raised in the callee.
The `invoke` instruction co-operates tightly with the `landingpad` instruction: the basic block that the exception branch of `invoke` points to must begin (after any possible φ instructions) with a `landingpad` instruction, and the entire basic block is called a *landing block*[31]. The `landingpad` instruction then encodes the list of exception handlers and whether there is a cleanup handler present, and which personality function to invoke for the corresponding callsite (`invoke` instruction). The syntax of the `landingpad` instruction is following:

```
<r> = landingpad <rt> personality <t> <pers_fn> <clause>+
<r> = landingpad <rt> personality <t> <pers_fn> cleanup <clause>*

<clause> := catch <type> <value>
<clause> := filter <array constant type> <array constant>
```

If the landing block is a cleanup one, the stack unwinder always transfers control to the landing block during the unwinding process, regardless of any exception handlers. If the landing block is not a cleanup landing block, it is only executed if some *catch clause* in the *landingpad* instruction matches the exception type (as decided by the provided personality function).[32]

[31] In upstream LLVM documentation, what we call a "landing block" here is referred to as a "landing pad". The reason for this departure is that the original terminology makes it easy to confuse "`landingpad`" as an instruction and "landing pad" as a basic block.

[32] Additionally, the filter clauses restrict the types of exceptions that can be propagated through the `invoke` instruction corresponding to this landing block, akin to how *exception specifiers* work in C++. If an exception is thrown and it reaches a filter clause of the appropriate type, a language-specific action is invoked. In C++, this action is user-specified, and defaults to terminating the program.

Since each *invoke* instruction only has a single landing block associated, this landing block is responsible for handling any and all *catch* clauses of the higher-level programming language covering the particular callsite. The return value of the `landingpad` instruction is crucial in deciding what action to take when the landing block is entered, and corresponds to the return value of the personality function. In other words, when the unwinder executes the personality function (which is part of the language runtime), it stores its return value, and provides this return value in the result of the `landingpad` instruction. Since the personality function has access to the part of the unwind tables generated from the `landingpad` instruction, it can communicate information encoded in the unwind table to the landing block itself. In the libc++ runtime, the personality function returns a tuple consisting of a pointer to the exception object itself, and a "handler switch value", an integer which corresponds to the index of a relevant "catch" clause of the `landingpad` instruction, or a special value (-1) when no catch clauses match but a cleanup needs to be performed.
The code generated for the landing block then checks the handler switch value computed by the personality function, and transfers control to a cleanup or handler block accordingly. Finally, if the selected handler is a cleanup handler, the exception propagation (stack unwinding) needs to be resumed after the cleanup is done. This is achieved by the `resume` instruction, which expects as a parameter the same value that was returned by the corresponding `landingpad` instruction which interrupted the exception propagation.
Interestingly, there are no LLVM instructions for *raising* (throwing) exceptions. This is left entirely in the management of the language runtime, which needs to closely co-operate with the stack unwinding library anyway (the interface of the personality function is mandated by the stack unwinder).

---

ex. 4.3 Consider the program in **Example 4.2**. We start in an out-of-memory condition in the program, at the point where `RC::RC()` is trying to allocate an array of characters. As a result, operator `new` throws an exception – the `throw` statement in the C++ source code of the implementation is translated to a `__cxa_throw` call. The `__cxa_throw` implementation then calls into `libunwind` – the `_Unwind_RaiseException` function in particular. At this point, `libunwind` takes over control, looping over active stack frames. Each frame is examined by calling the personality routine with a `_UA_SEARCH_PHASE` flag, in the context of the `throw` statement. In this phase, an exception handler is identified, but the stack is not yet unwound. In the next phase, the stack is actually unwound, and again, each frame is examined by a call to the personality routine. If a cleanup handler or the selected exception handler is found, it is invoked by returning `_URC_INSTALL_CONTEXT` to `libunwind` (otherwise, `_URC_CONTINUE_UNWIND` indicates that unwinding should continue with the next frame). Cleanup handlers return control to `libunwind` by invoking `_Unwind_Resume`.

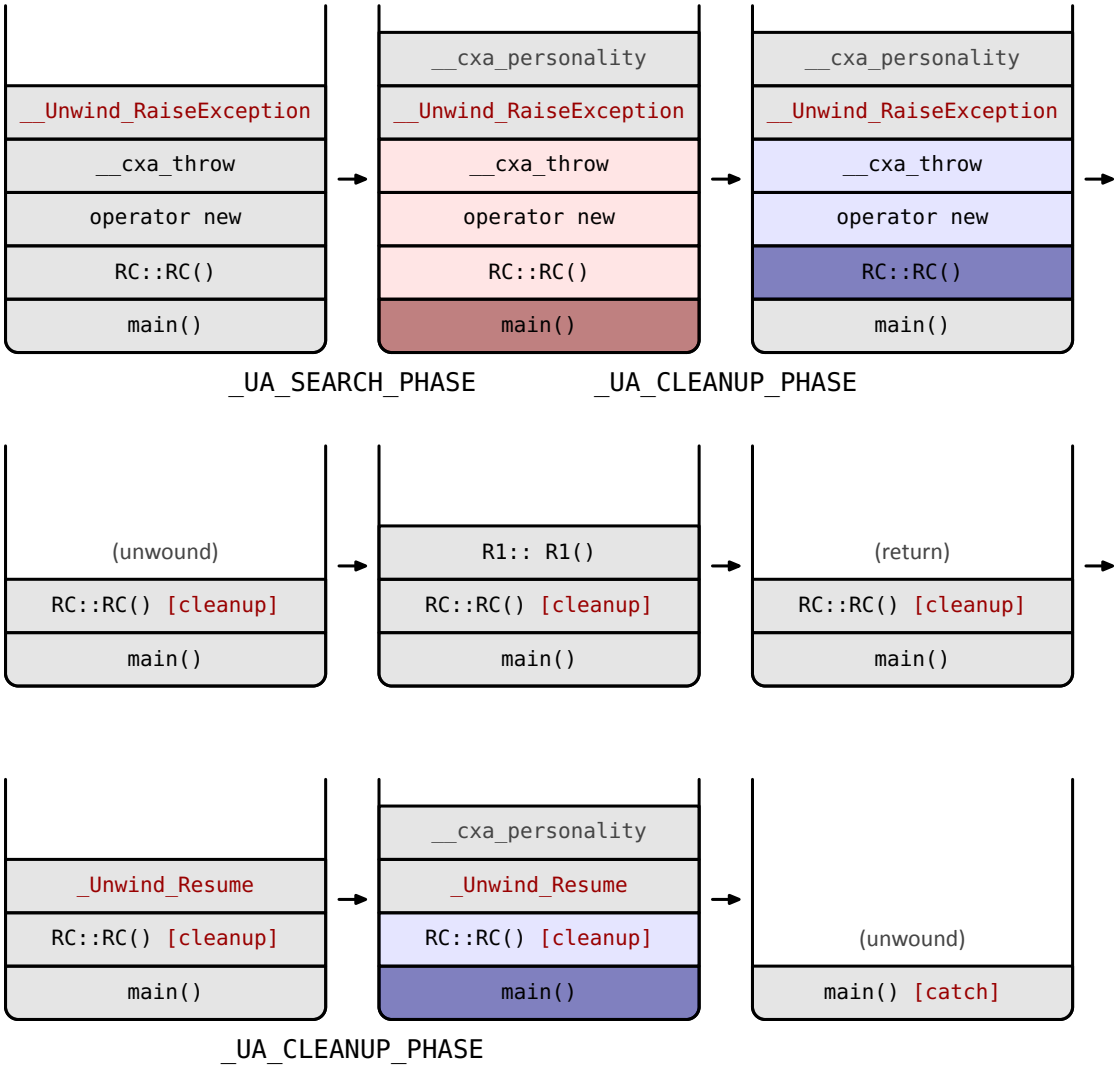


fig. 4.1 Execution flow from **Example 4.3**. Source code in **Example 4.2**.

4.1.6 Metadata

Since version 2.7, LLVM supports adding extensible metadata to the IR (and the metadata is persisted in the bitcode format). The purpose of this metadata is to allow specialised annotations to be added to the IR – by default, LLVM uses the metadata to encode debug information, which is later translated to a platform-specific format at the code generation time.
The metadata is formatted as a graph, where each node can contain a tuple of arbitrary LLVM values (integers, aggregates, etc.), in addition to references to other metadata nodes (which become the outgoing edges in the metadata graph). Metadata nodes can be either global, or they can be attached to instructions using a textual label (the label dbg, for example, is used by LLVM to provide location information for an instruction)

and they can be passed as parameters to intrinsic functions. They are, however, not first class LLVM values – they cannot be passed as parameters to regular instructions or regular functions. LLVM ensures that the metadata graph is properly de-duplicated: any two nodes that are equal are merged in the bitcode representation.
In addition to `dbg` metadata attached to individual instructions, it recognises calls to `llvm.dbg.declare`, an intrinsic which ties a variable declaration to the result of a particular invocation of `alloca`. Since various optimisation passes can move values from `alloca` memory to registers (and possibly remove the `alloca` calls entirely), LLVM also makes it possible to express value changes in user-level variables that are not currently tied to any particular memory location, using a call to `llvm.dbg.value`. Even though metadata is generally preserved by LLVM transformations (including optimisation passes), this is not a requirement, and in some cases metadata will be lost – especially in cases where instructions are removed by a transformation, metadata attached to those instructions will disappear as well.

4.2 Semantics

Since LLVM is an imperative language, small-step operational semantics are a good starting point. Rather luckily, LLVM IR does not have expressions: no intermediate, unnamed values arise, like they do in human-friendly programming languages. Each instruction is semantically atomic, and only refers to existing, named values. Additionally, each instruction changes at most a single component of the data portion of program state. As such, the semantic function is derived quite straightforwardly, by consulting the LLVM language reference [115] and almost mechanically translating the human-readable descriptions into derivation rules for the semantic function.
The program state (conventionally denoted σ when dealing with operational semantics) of an LLVM program consists of two interesting components: execution stacks (which in turn contain stack frames) and memory. In LLVM, execution stacks are conceptually separate from program memory, even though at actual execution time on most real-world machines, the stacks are implemented partly through memory and partly through CPU registers. Memory in LLVM is simple: the representation of memory in σ can be a fixed array of bytes, dictated by the size of the pointer type[33], effectively a huge n-tuple where n is the highest value a pointer can take. Only a small subset of instructions directly reference memory.

4.3 Control Flow

The graph induced by a single execution of a deterministic program is a linear sequence of states, with no branching (cf. **Section 2.2.2** and **Section 2.3**): each state has (at most) one successor. Each "edge" of this induced graph represents a single instruction and each node corresponds to a snapshot of the machine state visible to the program

[33] Pointer width is a fixed property and part of each LLVM program

(registers and mapped memory). In a sequential program, this "trace" is identical every time the program is executed with a given input. Without loss of generality, we can assume that input (and any interaction with the environment) is part of the program[34] (an assumption which is actually true in many interesting cases, notably various automated test cases, whether unit, functional or integration).
Generally, a trace that only has single instruction on each edge is more detailed than is useful. A chain of states can be collapsed if they are not relevant for analysis, forming a compound edge which represents an arbitrary instruction block. This technique is known as path compression [100 and 157].
However, while any single execution may yield a sequential trace, in parallel programs, the trace may be different every time the program is executed, due to non-determinism inherent in how instructions are scheduled by individual CPUs or cores, and a time-sharing, asynchronous nature of the entire system. This non-determinism is reflected in explicit-state model checkers by introducing branching into the execution trace (which is called a state space in this context), thereby encoding all possible interleavings. In any given state, the system makes a non-deterministic choice on which thread is executed next, creating a single successor state for each active thread. The number of states in the state space is exponential in the number of different threads.
While there are cases where different interleavings produce different end results, there are also many cases where the exact ordering of instructions is irrelevant: different interleavings will yield the same end state. Such confluent executions are redundant and only one of each equivalent set needs to be explored. This idea is at the heart of a class of techniques known as partial-order reductions [127].
In a state space (as opposed to a trace), path reduction can only straightforwardly apply to trace-like sequences of states, where each state has exactly one successor. However, such sub-traces do not naturally occur in state spaces of multi-threaded programs, since almost all states will have multiple successors caused by interleaving. Nevertheless, when a partial order reduction is applied, we choose a single execution among a set of many possible, replacing a diamond-like structure with a trace-like structure (cf. **Figure 2.2**) . This new trace-like structure is in turn amenable to path reduction, further reducing the number of intermediate states.
Both these reductions can be approximated statically, and one example of such an approximation is the τ-reduction [16], and a semi-dynamic τ+reduction [133]. More on these can be found in **Section 6.2**.

4.4 Heap

Most non-trivial programs nowadays use dynamic memory, also called a "heap". This memory is allocated on demand using function calls (usually `malloc` and its variants and `free`) provided by the runtime. The heap allows transparent re-use of memory

[34] There are other ways to efficiently deal with open-ended inputs and interactivity, most notably symbolic methods and abstraction. See also **Section 2.4** and **Chapter 7**.

that is no longer needed, without the requirement to allocate and de-allocate in first in / last out order like with the C-style stack.
We can consider a heap to be an oriented graph, with nodes representing individual objects and arrows representing pointers. A heap object is a result of a single allocation, it is internally always contiguous, but there is no guarantee on the actual layout of multiple objects in memory. In addition to pointers originating inside heap objects, there may be pointers in stack frames and registers pointing into heap objects (these are known as "root" pointers).

4.4.1 Pointer Tracking

For implementation of several desirable features in a model-checker, it is necessary to exactly know which pieces of memory are pointers, and where they point. The features in question are heap symmetry reduction (cf. **Section 6.3**), verification of memory safety properties (cf. **Section 5.1.3**) and tracking per-thread memory visibility (cf. **Section 4.4.2**).
One of the problems to solve is exact pointer tracking. While approximate solutions for C and C++ exist, these so-called *conservative* approaches [101] cannot be used for implementing heap reorganisation. A conservative collector will, in a nutshell, treat any bit-pattern as a pointer as long as it corresponds to a valid memory location. Since in a typical program, the heap size is much smaller than the address space and the heap is usually located near its end, this only introduces a small amount of harmless error for a mark&sweep collector, where in the worst case, some garbage is retained. However, a conservative collector must not alter pointers, since it could accidentally alter an integral value that has no relation to the heap, simply having the same bit pattern as a valid pointer.
This means that for successfully tracking pointers, we must use a tagging scheme, where an integer can never be constructed to resemble a pointer and vice versa. On one hand, shrinking pointers by one or two tag bits is not a problem – the address space of the model checker itself is a limiting factor, not the size of a pointer. On the other, it is not feasible to shrink integral types, as this would wreak havoc with established semantics of integer arithmetic[35]. Hence, we cannot easily prevent an integer from mimicking a bit pattern of a pointer. An alternative is to keep tagging information out of band, in a separate image of the address space. This is possible since we can instrument any and all memory access with updates to this tag space at the interpreter level.
All the tracked pointers are created in heap allocations, and their pointer status is preserved throughout their lifetime. We use a special pointer representation, where the heap object and offset into that object are kept apart and manipulated separately. This prevents pointers from overflowing into a neighbouring heap object (this would

[35] This scheme has been adopted in early garbage-collected runtimes, like that of LISP, where all scalars would reserve tagging bits and integer size would not match the machine word size. However, this approach is not feasible in low-level languages.

be a programming error, and must be detected) and makes pointer arithmetic safe and supported. Since programs may not make any assumptions about the bit content of heap pointers, they cannot be legally hijacked for integer constants. Therefore, we can safely rewrite the tracked pointers, without the risk of accidentally altering integral values, or missing actual valid pointers.
Finally, a simple yet efficient optimisation can theoretically further reduce the tracking overhead: since we can require and enforce alignment constraints on pointer values, any pointer value will start at a 4-divisible address, thus only requiring a single tracking bit per 4 bytes of memory.

4.4.2 Memory Visibility

The availability of exact pointer tracking offers an opportunity to improve τ+reduction. In its general form, τ+reduction operates with a notion of "observability": an instruction's effect is a cause for an interleaving point whenever this effect might have been observed by another thread. The main source of observability is writing to (shared) memory: in the thread-based programming model, all memory is implicitly available to all threads. However, it should be noted that in order for a thread to observe a memory write, it must be in possession of a pointer to that memory location.
Therefore, if a memory location has been allocated from the heap by a thread, but the pointer to this heap object is never provided to another thread, this memory location is essentially private to the allocating thread (this most importantly affects `alloca`-obtained memory – see also **Section 4.5.1**, although private heap-allocated structures are common as well). Since the layout of heap objects cannot be effectively predicted by the program being verified, it cannot "construct" pointers to objects out of thin air, and they must be explicitly shared by the allocating thread.
In order to effectively identify the area of memory visible to a particular set of threads (or processes), we trace the root set of these threads, the same way as a "mark" phase of a mark & sweep garbage collector.

---

ex. 4.4 We will demonstrate the idea on a simple program with linked lists. The layout of the program is shown in **Figure 4.2**. Three threads are currently running, all holding pointers into a shared linked list (these pointers are stored in the per-thread stacks).

4.5 Implementation

In order to enumerate the state space of an LLVM bitcode program, we need to be able to construct an initial state, and construct successors of any given state. The component of DIVINE that takes care of this task is called a *generator*, and in this case is implemented as a special-purpose interpreter (or equivalently, a virtual machine) which can efficiently store and load its state into a byte array (this array is called a *machine state vector*, and is discussed in detail in the next section).

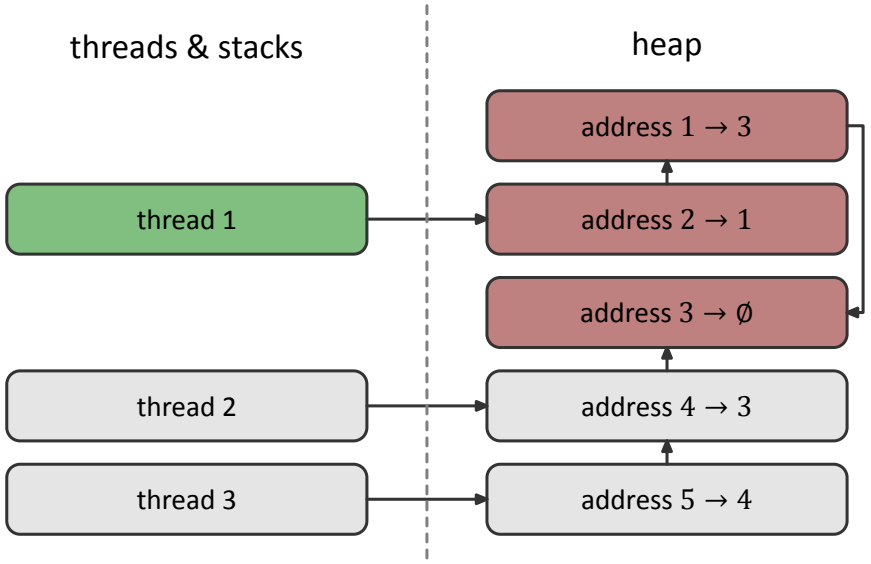


fig. 4.2 Memory visibility: Objects with red background are visible to thread 1, the rest is invisible.

In addition to the interpreter giving semantics to individual instructions, it also needs to provide an interface for operations that are normally not part of a CPU implementation but reside in the operating system: thread creation and scheduling, memory management, etc. This interface consists of a few functions (distinguished by their `__divine_` prefix), also called traps. We will discuss these in more detail in **Section 4.5.2**.
Also unlike a traditional interpreter or a virtual machine, a single instruction can have multiple different outcomes in our interpreter, causing multiple successors from a particular state (this is the case with a `call` instruction which refers to the `__divine_choice` trap; any other instruction will exhibit deterministic behaviour equivalent to what a normal CPU would do. However, there is another, more important, source of non-singleton successor sets, namely thread interleaving. When multiple threads are ready to execute, one is chosen non-deterministically to make progress.

4.5.1 Machine State Vector

An explicit-state model checker based on a virtual machine (like our LLVM interpreter), needs to be able to take snapshots of the machine's entire state in order to be able to explore the configuration graph (the state space). These snapshots should be compact and ideally stored as continuous, hashable blocks of memory. In an earlier version of the interpreter, the states needed to be unpacked into internal data structures and repacked every time a new snapshot was made. On the other hand, the current version takes a different approach, using the compact state representation directly to execute instructions, avoiding expensive unpack/repack operations. Moreover, since most of

the data required by the interpreter is packed close together in memory, its cache performance has improved substantially.

A running program on a contemporary commodity computer normally has access to a number of resources. The most important, apart from the CPU itself, is a bulk of random access memory that is traditionally divided into text (program), data, stack and heap. Most systems today, with only a handful of specialised exceptions, do not allow the text of a running program to be modified. In DIVINE, we treat it as constant. Apart from the program text, part of the data region of memory is constant and never modified by the program. This usually entails message strings and numeric constants used in the program. These two sections (text and constant data) are stored only once for each instance of the interpreter. The remainder is stored as a compact *machine state vector*, with layout illustrated by **Figure 4.3**.

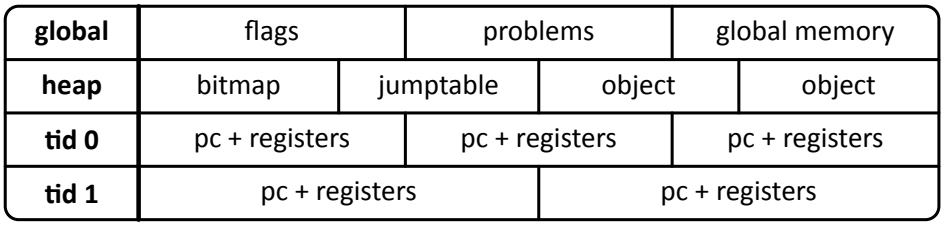


fig. 4.3 Representation of a state vector in a program with two running threads.

Out of the items in a machine state vector, the register stack needs special attention. Real machines (as opposed to virtual) have a limited set of registers, but a (comparatively) unlimited amount of memory. The "stack" in a C program consists of mapped memory and is used for many purposes: saving registers across function calls, storing return addresses and return values, and storing "automatic" local variables. All of this is organised into frames, and each frame on the C stack corresponds to a single entry into a C function.

Contrary to this, the LLVM virtual machine has an unlimited register file. When generating actual executable code, these virtual registers are allocated to machine registers and code for managing register spills (into the C stack) is inserted. However, at the level of LLVM instructions, access to the C stack is provided through the `alloca` instruction and is needed because values stored in registers have no address, and therefore cannot be passed by reference[36].

In our interpreter, we have a structure analogous to C frames, but our frames are not located in memory: they only contain register values and are not addressable (from the point of view of the code being executed). Since LLVM gives no guarantees about layout of memory coming from multiple `alloca` instructions, we allocate `alloca` memory from heap, which in our case is managed automatically. Therefore pointers to `alloca` memory go out of scope when their owner function returns and the heap memory

[36] Moreover, until recently, LLVM registers could not hold non-scalar values and those had to be stored in `alloca` or heap memory.

is freed. Due to the limited scope of alloca memory, the memory is explicitly freed (and as such, made inaccessible) whenever the owning function returns: this prevents programs from invoking undefined behaviour if a function returns a pointer to a local variable or stores it in a global variable.
As explained in **Section 4.4.1**, the LLVM interpreter needs to track which bytes in memory represent pointers and which do not. Additionally, it is useful to be able to discern uninitialised memory (memory locations obtained through allocation that have not yet been written to) from initialised (see also **Section 5.1.3**). This means that each memory location could be in one of three states: uninitialised, data or a pointer to a heap object. This information is stored separately from the actual memory, in bitmaps at the start or at the end of the corresponding memory or register area (the same tracking requirements apply to registers, and therefore to stack frames – a register may contain a pointer to the heap, or it may contain a value that is the result of a `load` instruction from an uninitialised memory location).
In addition to those memory state bitmaps (which exist for stack frames and for both global and heap variables), the heap needs to remember object boundaries. This is achieved through a "jump table", a vector of offsets into a contiguous memory area. Each entry in the jump table corresponds either to a start of a heap object or to the end of the heap. Size of each heap object is then obtained as a difference between two consecutive jump offsets. The representation of pointers used by the LLVM interpreter ensures that access cannot overflow from one object to a one stored in adjacent physical locations.

4.5.2 System Space

Our LLVM interpreter makes a clean separation between "system space" (the interpreter itself and whatever built-in functions – traps – it provides) and "user space" (the user code to be checked and any libraries it links, some possibly provided by DIVINE as replacements for system libraries). The separation within user-space is provided through linking – the implementation details of the DIVINE-provided library substitutions are opaque and they do not leak into user-supplied code.
An efficient user/system-space separation is in part facilitated by atomicity control provided through these three traps: `__divine_interrupt_mask`, `__divine_interrupt_unmask` and `__divine_interrupt`. These traps expose a low-level interface to atomicity control, making user-space implementation of library functionality much more feasible. When interrupt masking is in effect, the running thread must not be interrupted by any other thread, until after the masking is lifted. Moreover, the masking is bound to stack frames, which means that there is no danger of leaking the masking into user code, since a `ret` instruction to an originally unmasked function will automatically cause an unmask.
The advantage of explicit atomicity control is twofold: first, it makes library implementation much easier by avoiding the usual pitfalls of writing thread-safe code. Second, it substantially reduces the model checking overhead, since atomicized code is much

cheaper to execute, as no intermediate states need to be created. This effect is exponential, since every interleaving point in a library function essentially multiplies the number of states stored during its execution.
Apart from the three traps mentioned above, a small set of traps is provided, roughly falling into these four categories:

1. memory management:
   - `__divine_malloc` – obtain fresh memory from the heap,
   - `__divine_free` – force invalidation of all pointers to an area of memory,
   - `__divine_memcpy` – atomically copy a block of memory,
2. thread management:
   - `__divine_new_thread` – create a new thread, with a supplied function as an entry point and a pointer-sized argument
   - `__divine_get_tid` – obtain an identifier of the calling thread,
3. property specification:
   - `__divine_assert` – ensure that a value is non-zero
   - `__divine_ap` – insert an atomic proposition for LTL model checking.
4. control flow:
   - `__divine_choice` – non-deterministic (possibly probabilistic) choice,
   - `__divine_va_start` – obtain a pointer to the vararg block of memory,
   - `__divine_unwind` – unwind the stack
   - `__divine_landingpad` – obtain information about in-stack landing pads

4.5.3 User Space

Everything that is not system space is part of the user space, most importantly the language runtime, wrappers for DIVINE-specific functionality and the program itself. Everything that can be reasonably efficiently implemented in user space should be: functionality that the system space runs on behalf of the user space runs natively, as part of DIVINE itself. Hence, system space code is not subject to the stringent consistency and correctness checks the LLVM interpreter performs on each instruction executed in user space. Even if a component shipped with DIVINE is faulty, if it is part of the user space, any errors in that component will result in counterexamples pointing at the faulty component. In other words, a faulty user-space component can cause a spurious counterexample to appear. In contrast, faulty system-space components can compromise the model checking results entirely – since they have full access to the entire state vector, they can alter state of the program, and in case of grave bugs, even corrupt the memory of the model checker. As a result, faulty system-space code could even cause positive verification results despite presence of actual property violations. As such, it is desirable to reduce the size and complexity of the system space as much as possible, in effect reducing the likelihood of such problems.

ex. 4.5 To illustrate how the DIVINE system interface works, let us have a look at a small support function from the user-space `pthread` code.

```
void _cancel() {
    int ltid = __divine_get_tid();
    threads[ltid]->sleeping = false;

    // call all cleanup handlers
    _cleanup();
    __divine_unwind( 1, NULL );
}
```

This function implements cancellation of a running thread, and works in terms of system-space traps. The code is compiled into LLVM bitcode and linked into the user program as a library.

The atomicity control builtins outlined in the previous section play a crucial role in this endeavour. Without these, many low-level routines provided in the user space would be extremely inefficient, and in order to make model checking feasible, would need to be made part of the system space, with all the implied adverse effects. Main examples of this can be found in the `pthread` implementation shipped with DIVINE, which nowadays resides completely in user space, implemented in C++ using only the builtins enumerated above (a sample function from the `pthread` implementation is shown in **Example 4.5**). The same is true of stack unwinding support code, which is required for exception handling in many languages (we will discuss exceptions in the context of LLVM in **Section 4.1.4**, and in the context of model checking and DIVINE in **Section 4.6**).

4.5.4 C and C++ Runtime Support

While verification of LLVM bitcode is in itself an interesting theoretical feature, alone, it can't be fully used in practice, as very few programs if any will be written directly in the LLVM bitcode. If given a pure LLVM bitcode interpreter and a program in C, the only restriction on the program will be that it must not call any functions not defined in the program itself. In other words, C programs need no runtime support for language features. They usually do rely on a library of functions, which enable them to interact with the world. This library is usually called `libc` and is part of the operating system. In order to make verification of real-world code easier, DIVINE provides an implementation of `libc`, in form of bitcode that can be linked to (incomplete) bitcode produced by the compiler from the C program itself. While the implementation of `libc` is mostly complete, in some respects, it behaves differently from traditional OS-provided versions. Since the program that is being verified is not allowed to actually interact with

the world, such function calls are implemented either as "stubs" possibly using non-deterministic choice, or they interact with DIVINE using the system interface described in **Section 4.5.2**.

The case of C++ is slightly more complicated. While many language features require no special runtime support (i.e. the same as C), there are some that do, most notably Run-Time Type Identification (RTTI) and exception handling. Besides those areas where library support code is required for *language* features, like in C, most C++ programs make use of a standard C++ library.

Consequently, there are two libraries that are required by virtually all C++ programs: the runtime support library, and the standard library. Multiple implementations of both exist[37] – DIVINE ships with `libc++abi` for the runtime portion and `libc++` for the stdlib portion.

As far as RTTI goes, there are no special consideration with regards to model checking. The upstream `libc++abi` code can be used verbatim with DIVINE. Exceptions are more complicated, and are, coincidentally, a feature that is most often neglected in analysis tools and model checkers that work with C++ programs. Exception handling in C++ consists of three major parts: unwind tables, landing pads and exception handlers which are all generated by the compiler based on the input code, using special (although language-neutral) LLVM instructions: `invoke` and `landingpad` being the two most notable. Additionally, the C++ runtime library uses a CPU- and platform-specific *stack unwinder* and contains a language-specific *personality routine*. The personality routine makes use of the unwind tables generated by the compiler to guide the stack unwinder during an exception (see **Section 4.1.4** for details).

An LLVM interpreter hence needs to provide a stack unwinder and an API to access the unwind tables, for use by the personality routine. In DIVINE, the unwinder interface is extremely simple, consisting of a single trap, `__divine_unwind`. The language runtime can use `__divine_unwind` to remove a number of topmost stack frames from the stack of the current thread, returning control to the topmost remaining frame. If the active instruction in the target frame is an `invoke` instruction, control is transferred to its alternate destination basic block (a landing block), and the value passed to `__divine_unwind` is passed on to the personality routine of the landing block.[38] We will discuss exception handling in more detail in **Section 4.6**.

[37] The GNU compilers ship with `libstdc++`, which contains, as a subproject a runtime support library `libsupc++`. Clang ships with `libc++`. Depending on platform, a choice of either `libc++abi` or `libcxxrt` is available for use with `libc++`. An independent implementation is available from Apache Software Foundation under the name `libcxx`. Multiple compilers ship yet different implementations.

[38] If the active instruction is, however, `call`, the value passed to `__divine_unwind` becomes the return value of the `call` instruction. A type mismatch between the value passed and value expected is a fatal error and will be reported as a goal state by DIVINE. The personality routine however never unwinds to a frame which does not have an active `invoke` instruction.

4.5.5 POSIX Threads

Another area where special attention to user space is needed is in the area of thread management. Neither C nor C++ offer a native concept of threads in the language itself, and LLVM follows this trend. In the case of C++11 (and to a lesser degree, C11), threading primitives have become a standard part of the runtime library. Nevertheless, a different, language-independent interface for thread support predates both these by a decade, namely that of POSIX.1-2001, also known as `pthreads`. In practice, both C11 and C++11 runtime libraries are often implemented in terms of `pthreads`, and are otherwise both platform and CPU independent. Since vast majority of parallel programs written in C and C++ still uses the classical `pthreads` interface and since both C11 and C++11 runtime libraries have implementations in terms of `pthreads` available, it is only natural to provide threading primitives in form of `pthreads`-compatible API. Hence, DIVINE is shipped with a bitcode implementation of most of the `pthreads` library, built on top of the few thread-control and atomicity-control traps. This way, programs using either the traditional `pthreads` API, or ones based on C11 or C++11 threading primitives can be easily verified.[39]

4.5.6 Other Languages

While DIVINE itself provides runtime support for C and C++ programs in form of bitcode libraries, this is not the case with runtimes for other languages that can be compiled into LLVM bitcode. For these languages, in order to verify programs, it may be necessary to provide bitcode for the language's runtime, possibly modified in ways similar to our C and C++ runtimes discussed in previous section. In many cases, the runtime will be based on the C library and require only minimal modifications. The components most likely to require further attention are IO and exception handling (also depending on how are exceptions translated by the language's compiler into LLVM bitcode).

4.6 Exception Handling

We have outlined the mechanisms used by LLVM to implement language-agnostic exception handling in **Section 4.1.4** and **Section 4.1.5**. There are multiple points where DIVINE has to hook into those mechanisms in order to support exception handling in a particular programming language. While a substantial part of that support is language-agnostic, crucial pieces of infrastructure are part of the language's standard library: in case of C++, this is `libc++abi` as explained in **Section 4.5.4**.

[39] Unfortunately, the C++11 threading API introduces significant overhead, since its verbatim implementation from `libc++` is used in DIVINE, without introducing any atomic sections. Interestingly though, we have discovered a crash bug in the implementation of `std::thread` constructor, cf. **http://llvm.org/bugs/show_bug.cgi?id=15638**.

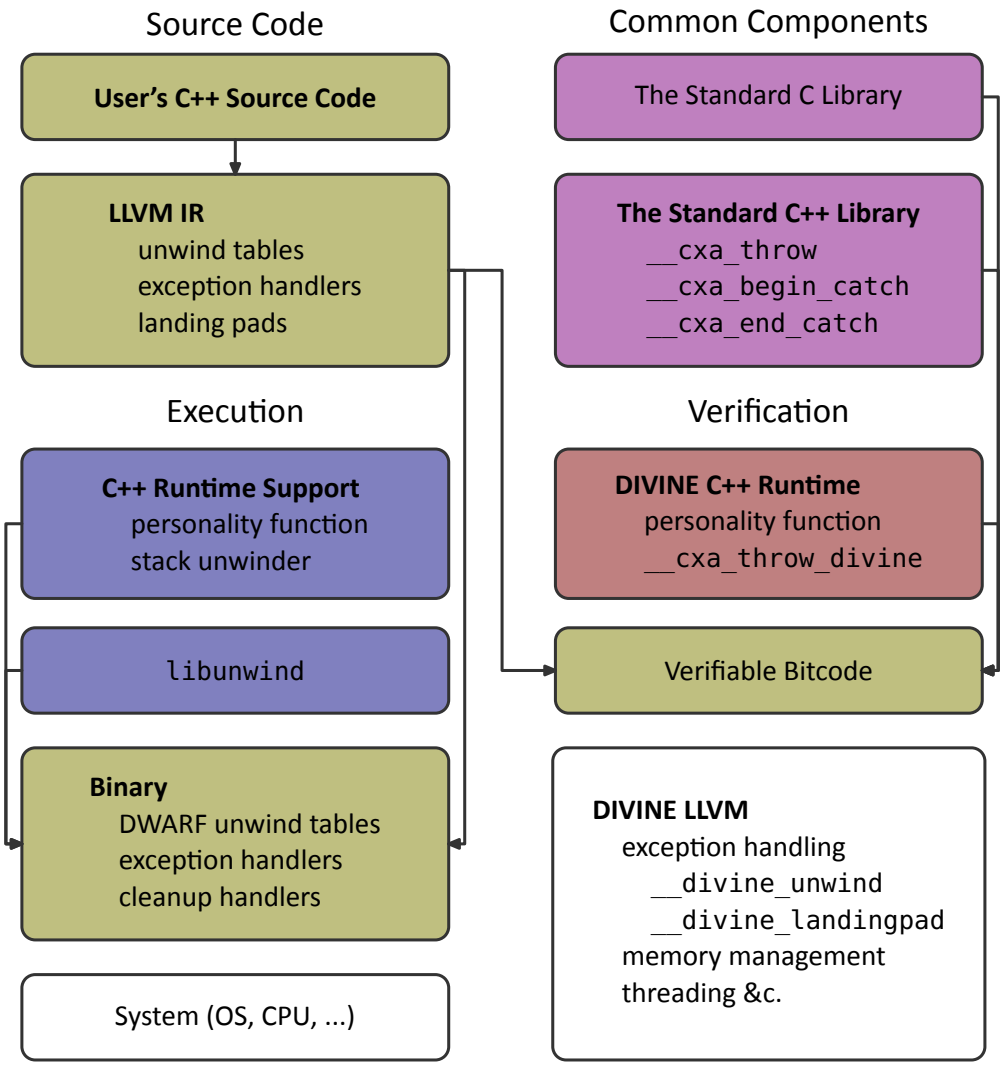


fig. 4.4 Components involved in exception handling.

In a native code generator in LLVM, the information from `landingpad` instructions generated in the frontend is used to construct unwind tables. The format of those tables is platform- and architecture- specific. To read those tables, `libc++abi` uses the `libunwind` interface (originally specified as part of the IA64 C++ ABI). This interface is semi-standard, but no actual standardising document exists. Since the `libunwind` implementation is tied to the binary format of the executable, via the in-memory image of the unwind tables, it cannot be directly used in DIVINE. Likewise, it is tied to a specific architecture/platform via its knowledge of stack and register layout – another disqualifying feature. Therefore, `libunwind` needed to be replaced with a new implementation for DIVINE.

ex. 4.6 There is a number of components involved in exception handling (shown in **Figure 4.4**) that interact with execution and/or verification. The source code is first compiled using a suitable C++ frontend (clang or gcc) into LLVM IR. When building a binary for execution, the IR code is fed to a code generator and combined with common components (the standard C and C++ libraries), and with execution-specific components: `libunwind` and execution-specific parts of the C++ runtime support library (the personality routine and the `libunwind`-based stack unwinder). For verification purposes, the LLVM IR is instead combined with those same common components that have been converted into intermediate representation, and with verification-specific runtime functions from the DIVINE C++ runtime. The resulting bitcode file is then fed to DIVINE, using its LLVM subsystem to generate the state space and execute a suitable verification algorithm on that state space.

4.6.1 The `libunwind` interface

There were two basic options: the first was to replicate the portion of the `libunwind` interface used by `libc++abi`, making it possible to use unmodified source for `libc++abi` – which sits on a higher level than `libunwind`. Conceptually, this is a tempting solution – the more of the library code is left intact, the more faithful the verification. There is a major downside though: the interface between `libunwind` and `libc++abi` is complex and intricate. This is especially true of the interface between the unwinder and the personality function: the unwinder uses the personality function as a callback, invoking it once for each active frame on the stack at the moment exception is raised. The personality function uses a pair of platform-specific registers to pass the handler switch value and the exception pointer to the exception handler: it cannot invoke the handler itself, as the stack has not been unwound yet and the handler would end up running in the wrong context. For this reason, `libunwind` provides an interface to splice register values into the context of the exception handler to be invoked.[40] It would be in principle possible to implement this interface in DIVINE system space: each thread would need two special thread-local variables to hold these values, and the `landingpad` instruction would simply read those values and copy them into appropriate LLVM registers. The downside is extra space overhead – 16 bytes per thread, allocated even if no exceptions are currently active.[41] Another downside is that this limits flexibility: while the LLVM exception mechanism is made to play nice with `libunwind`, it is flexible enough, at

[40] This is clearly implemented in a platform-specific fashion. If the registers are always saved on the stack, their stack images will be rewritten. If they are clobber-type registers, they can be written to directly and the unwinder will take care not to clobber them before transferring control to the selected exception handler. Other options may be available depending on platform.

[41] Those 16 bytes could be compressed away in most cases to a single bit, at expense of code complexity. However, system-space complexity is very costly, and complexity involved in addressing the state vector even more so.

least in theory, to admit another approach to stack unwinding. Using this approach would mean changing the DIVINE system space to accommodate a different `landingpad` return type.
While this API/ABI issue can be reasonably solved, there is a more important issue at play. Even though `libunwind` understands the platform-specific portions of unwind tables, it provides no support for parsing the language-specific chunks. This means that `libc++abi` code itself has ABI-specific knowledge of the unwind table layout, needed to extract the exception type info and switch values. All `libunwind` does here is provide a pointer to the `lsda` (language-specific data area) portion of the unwind table for a given stack frame. In order to support this `libc++abi` code in its literal form, DIVINE would have to synthesise DWARF-formatted[42] `lsda` areas from `landingpad` instructions. This is unpleasant, because it is a complex format designed for space efficiency, and the encoded tables are completely C++ specific, even specific to C++ on a particular platform. The only reasonable way to provide such tables would be to leverage pieces of the existing x86 (or x86-64) code generator to synthesise the `lsda` tables. LLVM, however, does not provide an interface to this functionality.

4.6.2 DIVINE-specific unwinding API

Both these issues in mind, we have chosen a different approach, which requires modifications to `libc++abi`, but can be implemented with just 2 new system-space builtins – one for querying metadata encoded in `landingpad` instructions, based on a stack frame reference (`__divine_landingpad`) and another for actually unwinding the stack (`__divine_unwind`).

---

ex. 4.7 Let us now consider the exception-handling process as it happens in the DIVINE runtime (see **Example 4.2** for the source code and **Figure 4.5** for a graphical depiction). The situation at the start corresponds to an out-of-memory condition in the program. Constructor of class `RC` was trying to obtain dynamic memory (using operator `new`), but the allocation request has failed. As a result, operator `new` is throwing an exception – the `throw` statement in the C++ source code of the implementation is translated to a `__cxa_throw` call, which uses `__cxa_throw_divine` to unwind the stack. The unwinder first uses `__divine_landingpad` to find an exception handler (which it finds in the call frame of the `main()` function, and any intervening cleanup handlers (there is one in the `RC` constructor itself). The unwinder proceeds to call the personality routine to obtain a handler switch value and passes the result to `__divine_unwind`, along with the address of the first cleanup handler. `__divine_unwind` removes stack frames up to the cleanup handler, which takes control and calls a destructor of the locally constructed `R1` instance. Finally, when done, the cleanup handler invokes the `resume` instruction

[42] DWARF is a companion format to encode debug and other metadata in ELF executable images. A backronym "Debugging With Attributed Record Format" has been invented for it.

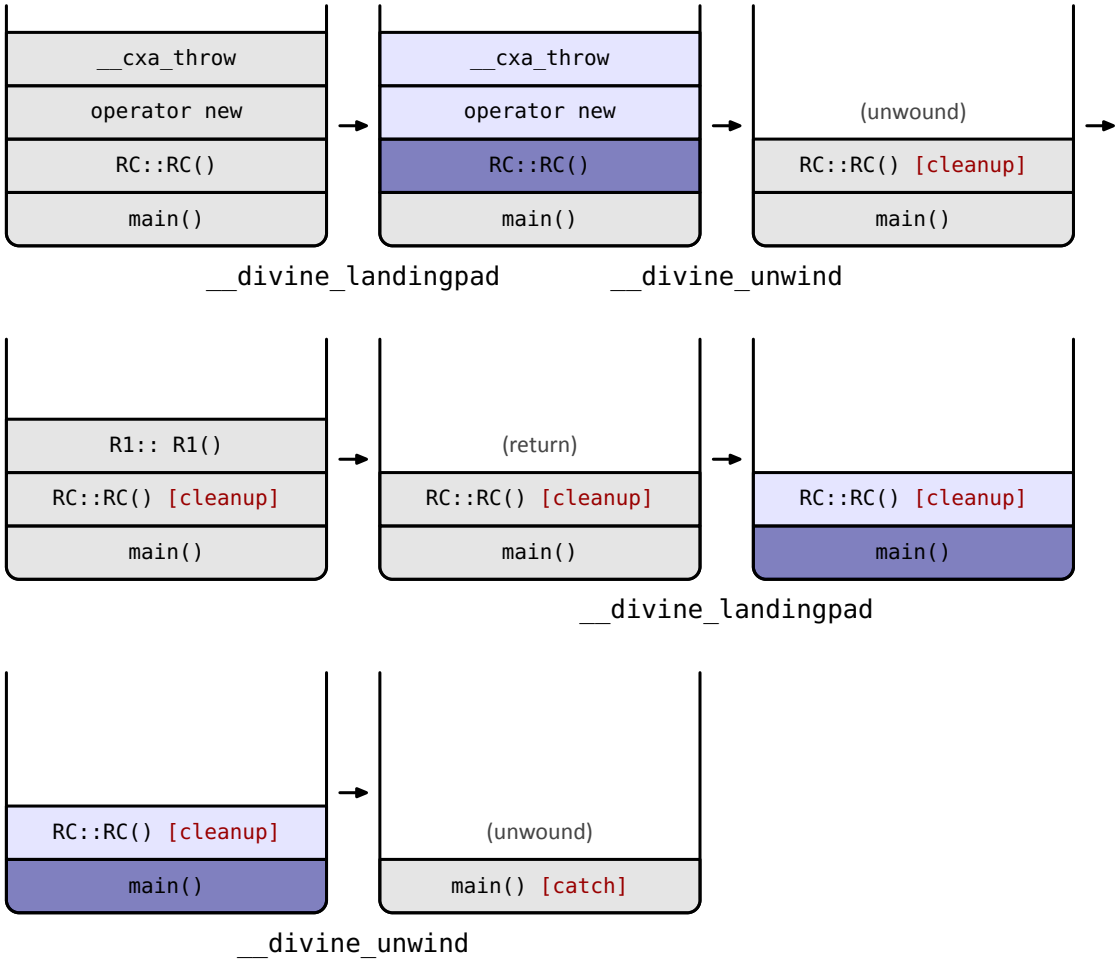


fig. 4.5 Execution flow from **Example 4.7**. Source code in **Example 4.2**.

which continues the propagation up the stack, to the exception handler (the catch block in `main()`).

This clearly requires some changes in `libc++abi`: one is the personality function, and the other is the actual `__cxa_throw` implementation: a call to this function is inserted by the C++ compiler at the site of a `throw` statement (along with some support code). While in the original `libc++abi` implementation, the personality function bears most of the burden (since `libunwind` does the stack search, calling out to the personality function as needed), this is reversed in the DIVINE implementation. Here, the personality function merely extracts the correct items from the exception header to pass on to the exception handler. The `__cxa_throw` implementation, on the other hand (and unlike in the `libunwind` version), unwinds the stack itself, using `__divine_landingpad`. This builtin does not change anything, but provides the caller with `landingpad` metadata, using a simple integer indexing of stack frames. Negative indices start at the top of the stack, non-negative at the bottom. This makes it easy for the unwinder to walk

through the stack one frame at a time, looking for an appropriate handler. When the handler is found, it can call the personality function (pointer to which is part of the `landingpad` metadata) and pass it to `__divine_unwind` along with the frame address it obtained from calling `__divine_landingpad`. The job of `__divine_unwind` is then simple enough: destroy all the frames above the one addressed and transfer control to the landing block associated with the active `invoke` instruction in the now-topmost stack frame. `__divine_unwind` also takes care of copying the value it obtained from its caller (in this case the return value of the personality function) into the result of the corresponding `landingpad` instruction.
The implementation of `__divine_landingpad` takes advantage of the implicit garbage collection done by DIVINE, as it allocates the metadata block on heap. Since the block is neither flagged as a result of an `alloca` instruction, nor as a result of `__divine_malloc`, it is transparently retained as long as necessary without being flagged by the interpreter as a memory leak.

4.6.3 `setjmp` and `longjmp`

The C functions `setjmp` and `longjmp` can be used for non-local transfer of control, in a way similar to C++ exceptions. In fact, some C programs use those two semi-standard functions to implement somewhat crude exception handling in C. The purpose of the `setjmp` function is to set save enough of the machine state to allow non-local transfer of control to the point in program where `setjmp` was called. The `longjmp` partner then, using a context saved by the `setjmp` call, restores the corresponding machine state. The state is exactly the same as it was right after `setjmp` call returned for the first time, with one exception: the return value of the `setjmp` call is altered in its second return, to make it possible to detect whether the return was a "normal" return or a `longjmp` return.
Clearly, exception handling based on `setjmp`/`longjmp` cannot be "zero-cost" – state has to be explicitly saved at the start of every `try` block, and possibly before any resource acquisition. The latter problem can be side-stepped by maintaining a separate "resource" stack [148], but even then, entering `try` blocks is fairly expensive. Nevertheless, robust C programs may choose this style of exception handling, since the runtime overhead can be outweighed by the programming benefits – especially due to fewer and simpler error paths to write, maintain and test. Finally, there are other uses for `longjmp` in programs, besides exceptional situations.
While `longjmp` is not nearly as widely used as C++ exceptions are, the reasons for supporting this primitive are similar, even if somewhat weaker. Fortunately, the primitives we have designed for C++ exception handling can be easily re-used in implementing `setjmp` and `longjmp` – since `__divine_unwind` can just as easily stop at a `call` instruction as it can on an `invoke` instruction, we only need minor extensions to the `__divine_landingpad`/`__divine_unwind` mechanism. The main difference between exceptions and `longjmp` is how the control flow at the point of `setjmp` is handled. The DIVINE-specific implementation of `setjmp` needs to be able to find out the program

counter value of its enclosing frame, corresponding to the call instruction. This can be done by slightly extending `__divine_landingpad`, to provide the program counter value for `call` instructions in the stack (this does not alter the semantics of `__divine_landingpad` for `invoke` instructions in any way).
Finally, `__divine_unwind` needs to be extended as well, to allow the caller to specify where to restart the execution in the target frame – since `longjmp` is not above the corresponding `setjmp` in the call stack, a successful `longjmp` needs to change the program counter in the target frame, in addition to unwinding. Luckily, this is fairly easy, since the `__divine_unwind` caller can specify the program counter corresponding to the callsite to unwind to. For normal C++ exceptions, the caller just puts in a 0, meaning no program counter adjustment (i.e. the semantics stay exactly the same) and `longjmp` passes in the program counter value obtained from a `__divine_landingpad` call done by the `setjmp` function.

4.6.4 Use Cases

Besides the simple fact of making model checking possible on a substantially wider class of programs, exceptions themselves are an interesting subject for model checkers: error paths are notoriously hard to test. With a model checker, however, it is easy to insert non-deterministic failures and check that the program behaves sensibly under all sorts of error conditions. Resource leaks are among the most common errors encountered in error paths, which makes the problem even harder to debug – resource leaks, especially memory leaks, require special tools to diagnose in a test, such as valgrind.
Since DIVINE can already diagnose memory leaks in LLVM inputs, checking error paths involving exceptions becomes a fairly easy task. However, error paths can contain more serious errors as well – especially in multi-threaded programs, where threads are not isolated from the effects of other threads failing to handle an exception, and the entire program may crash. Among the first issues that we have found using our new exception support in DIVINE is such a crash, in `std::thread` implementation in `libc++`, under out-of memory conditions.[43] When a new thread is created using this standard C++ interface, most of its state is allocated in the newly-created thread, before user code is executed. Since this allocation can fail with an exception, and the `libc++` implementation fails to install an exception handler in the context of this newly-created thread, the exception cannot be caught. In such cases, the C++ standard requires the runtime library to call an "unexpected exception handler", which, unless overriden by the user, terminates the application.
In order to fix this problem, we have moved the memory allocation code into the calling thread. To avoid synchronisation problems and possible resource leaks, this happens before the new thread is created – the calling thread allocates all the dynamic state for the new thread and passes it down as a parameter. This way, any exceptions related to

[43] The proposed patch that fixes the problem can be found at **http://llvm.org/bugs/show_bug.cgi?id=15638** and the relevant source code in the file `libc++/std/thread`.

resource exhaustion happen in the calling thread, in a context where users can control the scope and propagation of exceptions by wrapping the call to the thread constructor in a suitable `catch` block.

4.7 Counterexamples

While the main focus of this thesis is model checking, an extremely important and often underestimated ingredient to its practical usefulness is counterexample presentation. For the user, the counterexamples comprise the "language" which the model checker uses to convey new information it has gathered during the process of model checking. When all properties hold, the output is trivial: "yes". However, if the answer is "no", the bulk of the output describes "what went wrong". Outside of model checking, an extremely elaborate set of analysis tools exists to analyse program crashes and misbehaviours, usually under an umbrella of so-called "symbolic debuggers".

These tools provide many ways in which to explore and manipulate traces – stepping through the program, examining individual variables at a particular point in execution, altering variables to steer the program into a different code path, among others. Some of these options could be regarded as a poor man's substitute for model checking: especially manipulating scheduler behaviour by hand or tweaking variables in an already running program could be regarded that way. On the other hand, injecting behaviour changes into counter-examples and recomputing them could be a tactic useful also in conjunction with model checking.

In contrast to symbolic debuggers and their rich interface, most model checkers can only provide a textual trace of the counterexample. One could argue that most of the work is, in the case of model checking, done "up front", partly by changes to the program and partly by the model checker itself. However, part of the reason why debuggers are so successful is that they allow the user to dynamically focus on a particular aspect of a problem – show the evolution of a particular variable during a run, for example, or monitoring one variable and analysing a heap structure when a certain change to the monitored variable happens. In this sense, debuggers are primarily tools for extracting useful data about a program run efficiently.

If a model checker is limited to a textual trace, a trade-off must be made in which data about the program to include. If the entire state of the program is dumped after each instruction (as an extreme example), this easily leads to megabytes of text even for the most trivial traces. Clearly, such a tool cannot be used directly by a human and would require another layer of tools to assist with its interpretation. On the other hand, if some data is permanently elided, no matter how careful the heuristic for doing so, it is very likely that in some cases, an essential data point will be missing from a trace.

At the moment, DIVINE provides rich interactive simulation for DVE models (i.e. where the structure of the model – variables and processes – are fully exposed to the user), but not for LLVM programs. In the latter case, only a textual description of program states is available, with no further structure. The generic interactive simulator provided by DIVINE (the `simulate` subcommand) only works in terms of entire states: this is

adequate for programs with small and simple states, but as outlined earlier, this is insufficient for analysis of more complex situations.

4.7.1 Machine-Readable Counterexamples

Nevertheless, a full interactive environment for manipulating high-level programs[44] is out of scope of DIVINE itself, at least at the time being. However, it would be very helpful to facilitate development of a stand-alone tool, or alternatively an extension of an existing LLVM-aware symbolic debugger such as `lldb`, which could analyse traces generated by DIVINE.

[44] Here we mean programs in C or C++ and possibly other languages. While an LLVM-level system would be much easier to implement, it would be also only marginally useful to end users, who are generally interested in verifying C and C++ programs and use LLVM bitcode only as a means to this end. The relationship between the C source code and the compiler-generated LLVM bitcode is in general too tenuous to analyse by hand.

5 Property Specification

In order to perform verification, software or otherwise, besides the system to verify itself, another vital component needs to be provided: the specification. There are many ways to describe what a program should (or should not) do. Properties most widely checked on real software are in the form of safety assurances: certain bad things never happen. An assertion is never violated, the program never deadlocks, the program never dereferences a `NULL` pointer, divides by zero or writes into deallocated memory. Many of these properties are normally checked via testing, sometimes assisted by a runtime checker like valgrind [123]. In many cases, this is adequate with sequential programs. However, multi-threaded programs can exhibit non-deterministic behaviours, with some of the undesirable execution branches being extremely rare. Likewise, testing is less suitable for checking temporal liveness properties of reactive programs – expressing good things which should keep happening. Again, this is an area where model checking can be helpful.
We will discuss how DIVINE can help programmers establish some of these properties of their programs, and how to express them in a form suitable for DIVINE.

5.1 Safety

As outlined above, safety is among the most-often checked properties in software. It only serves as a reinforcement that it is also one of the easiest to check, whether via testing or with more rigorous methods. Safety of a program can often be demonstrated by a simple reachability analysis – enumerating all reachable states of the system and checking that each such state satisfies the requisite safety specification. This can be done explicitly, by enumerating the set of reachable states one by one, as in explicit-state model checkers, approximated by testing where certain key configurations of the system are examined, or symbolically by building appropriate formulas or another concise description of this set.

5.1.1 Assertion Safety

Many programming languages come with a builtin or a library routine called “assert”. The use of runtime assertions is multiple: they help programmers document pre- and post-conditions in their program in a machine-checkable form and they help with testing because testing builds of software usually include runtime code for checking such assertions whenever they appear in the source code.
When an assertion is detected to be violated at runtime (in a testing build), the program is forcibly terminated and the programmer can analyze program state in which the assertion was violated. Release builds are usually compiled with assertion checking disabled, and any assertion failures become silent, possibly resulting in later undefined and undesirable behaviour.

For a sequential program, running the program is often sufficient to determine whether all assertions hold up. In programs with race conditions, assertions may only be violated sometimes. DIVINE, performing reachability analysis on a program with assertions, uses the same *assert(bool)* construct as traditionally used in testing. Hence, verification of assertion safety is a matter of using assert statements of the programming language to express key pre-/post-conditions.

5.1.2 POSIX Deadlocks

Another common problem with multi-threaded programs is the possibility of a deadlock, a situation where all threads wait for each other. In model checking of more traditional asynchronous models, a deadlock is normally defined as a system state with no successors. However, this definition is not very useful for threaded programs if mutual exclusion, or any other synchronisation primitive is implemented in terms of spinlocks or some other mechanism which does not entirely block execution. With a spinlock, the thread waiting for the lock will run in an infinite loop trying to get the lock, forming a self-loop (or possibly a longer cycle, depending on the complexity of the locking protocol). Hence, the traditional approach of detecting successor-free states is inadequate for deadlock detection in threaded programs.
Instead, the threading runtime can track a list of threads waiting for each particular lock (or blocking resource in general). With this data, upon each lock request, the runtime can detect whether a deadlock has formed by examining the dependency structure of locks. A program is (partially) deadlocked whenever a cycle forms in this dependency structure: in other words, if thread A tries to obtain a lock L while it already holds lock M, thread B is waiting for the said lock M and already holds L, a dependency loop has formed and deadlock should be signalled. A program is fully deadlocked if the dependency cycle spans all threads.
The detection of these situations is entirely delegated into user space (cf. **Section 4.5.3**). This means that in order to take advantage of this deadlock detection scheme, programs need to use locking interfaces provided by the threading library shipped with DIVINE. Due to increasing popularity of atomic exchange-swap and read-modify-update instructions, it would be worthwhile to offer a more general mechanism for detecting non-progressing cycles in the state space, as these instructions can be easily used to implement special-purpose spinlocks in user-level code, bypassing the deadlock detection provided by pthread locks.

5.1.3 Memory Safety

Memory safety is a somewhat fuzzy category, and could be also thought of as *pointer* safety. We usually require that the program only reads and writes memory that has been properly allocated and initialised. The most common memory error is probably a null pointer dereference: when a pointer in the program takes a special “invalid” value (called a null pointer and *usually* implemented as an all-zeroes bit pattern) –

indicating that the pointer is not in use – but it still tries to access memory via this pointer. This usually comes about either by errors in program logic (a condition is thought sufficient to guarantee a non-null pointer but in fact is not) or by incomplete or erroneous handling of exceptional situations. The latter is prevalent due to the difficulty of testing just such cases – exceptional situations are, after all, a fairly rare occurrence. Another common problem is a dereferencing of a non-null but still invalid pointer. This might be because the pointer was to a dynamically allocated area on the heap which has been de-allocated in the meantime, but the pointer has failed to be updated, or might be due to a bound error – the pointer is indexed by a value outside the permissible range. Both these cases are often very hard to diagnose – unlike in the null pointer case, the program is not aware that the pointer is invalid at the point of use, and undefined values may be read – or worse yet, unrelated memory may be overwritten, leading to a crash or other bad behaviour much later in the program.

note A special class of debuggers exists to diagnose this kind of problems. The program is thoroughly annotated, or even interpreted (as is the case with valgrind, probably the best-known such debugger). All memory access is checked at runtime for boundary conditions and memory re-use is limited to facilitate diagnosis of access into freed heap memory.

In practice, it is very hard to exhaustively check this class of errors, even with rigorous automated tools. It is very hard to prevent the program from "accidentally" constructing a valid pointer that nevertheless points to an unexpected memory location. Since *some* pointers must be valid, and as long as pointers are expressed as numbers, a difference between two valid pointers (into unrelated memory areas) will always yield an out-of-bounds index for one of them that will, upon use, lead to the second valid pointer. If values are known with certainty, at runtime, to be (or not be) pointers, it is possible to guard arithmetic operations on those against this particular problem.

note It is, however, not sufficient to prevent operations that create out-of-bounds pointers entirely. The program may (and real programs often do) store a pointer just past the valid region of memory, to serve as an access sentinel. The way this is usually handled is by creating *red zones* of addresses on both sides of valid memory objects. Within the range of the red zone, pointer construction is admitted – but pointer dereference is not. If an arithmetic instruction was to push a pointer beyond this red zone, overflowing into a valid memory location, this would very likely indicate an error (as long as the red zone is sufficiently large). In fact, even a "false" positive in such a case may be valuable, as such behaviour is rarely intended even if technically correct.

5.1.4 Memory Leaks

A distinct memory problem, while technically a safety issue, is closely related to liveness – it does not directly pose a safety violation, as the program will continue to run as normal. It does, however, threaten the continued functioning of the program: if memory is allocated but never freed, it will eventually run out. This can happen independently

of memory leaks: the program could just accumulate working memory, retaining references to all allocated blocks. However, if all references to a particular piece of memory are lost, the memory cannot be reclaimed anymore. It is this irreversible loss of memory that is usually called a "memory leak" and is what is diagnosed by most memory debugging tools.
For detecting most straightforward memory leaks, existing runtime- and testing- based tools are quite adequate. The area where model checking can substantially contribute is memory leaks that arise due to exceptional situations in a program. The error paths in most programs are extremely sparsely tested, since basic testing tools do not offer ways to synthesise program failures, and they only rarely happen spontaneously. Hence, it is common for programs to leak memory under various error conditions. In many programs, this is not a huge problem: since error paths are rarely invoked, the amount of memory that can be lost is fairly limited. However, the effect is cumulative in a long-running process, such as a server or a daemon. In an environment with constrained memory, it could easily become fatal.
Since DIVINE already needs to perform heap canonisation, detecting memory leaks in the process is quite simple: instead of simply discarding unreferenced memory, it is instead attached to a problem report, and the state where this has happened is flagged as erroneous.

5.1.5 Points-To Safety

The "safety" of points-to information is a somewhat technical property, mostly useful in program transformation and not so much of direct benefit to programming in general. The idea is that many program analyses need accurate pointer information to function properly (cf. **Section 7.6**). While efficiently approximating this information is difficult, it is fairly easy to verify – as an aid in implementing algorithms for pointer analysis, DIVINE provides means to check that the pointer information embedded in LLVM bitcode is accurate.

note The implementation of points-to safety in DIVINE requires that the pointer analysis in question uses a particular persistent way to represent its results. The particular format for LLVM metadata is described in **Section 7.6.3**.

5.2 LTL

Normally, an LTL formula in a specification means "all possible runs of the system satisfy this formula" – the formula itself is interpreted in terms of system runs (cf. **Section 2.8**). A run consists of a sequence of states, and in state-based LTL, the atoms of the formula refer to such individual states of the system. The usual definition given when establishing LTL semantics is to suppose a function $L : S \to 2^A$ where S is the set of system states and A is an arbitrary finite set of atoms. The LTL formula then simply refers to elements of A and the actual evaluation (or model checking) of the formula "looks" at the states through an L-mapping.

Then, the user is usually allowed to specify L using an expression language of arithmetic, boolean and relational operators (cf. **Section 2.8.1**). These expressions are evaluated in a sort of "global scope" of the model – this is facilitated by the simplistic, mostly fixed form of a system state in a typical modelling language. Often, there is a simple addressing mechanism for processes (trivial in case of languages with a fixed number of processes), and each process has a simple flat variable scope – in other words, each process has access to a fixed set of variables throughout its lifetime. Usually, individual "control locations" of a process can be labelled using names and addressed in a similar fashion. Finally, the shared global variables live in a flat namespace available to all processes in the system, again of fixed size and layout.
Clearly, a language with dynamic memory and arbitrary scope nesting makes such simplistic implementation of L quite impossible. No static scope can be defined that would serve for evaluating expressions: values have limited lifetimes and an expression referring to a currently undefined variable would itself become undefined. This is a major problem, even if we ignore the question of a suitable variable addressing scheme. In theory this could be overcome by introducing a "defined?" predicate, and require each variable use to be guarded. However, this would obviously become extremely unwieldy and useless to give succinct and clear property definitions. An intricate and complex variable addressing scheme would further reduce usability of such a solution.

5.3 Atomic Propositions

Therefore, we need an alternative approach for describing L. Since L is a function, it is highly desirable that the language used for its definition be referentially transparent. Implementation of L that would fail to meet this requirement would cause LTL model checking to give incorrect and incoherent results.
Overall, there are two basic approaches that can be taken. We can either make the language extremely simple, basically just adding a constant-sized block of boolean flags to each state, and allowing only boolean expressions over these flags. The flags then need to be manipulated from within the system to facilitate LTL model checking. The overall expressive power however seems to be rather unsatisfactory. The other option is to make the language substantially more powerful. A language with access to complex (recursive) data types and to (at least structural) recursion would be able to process the heap and stack using graph and list maps and folds. We will present such a language in following sections.

5.3.1 The SILK Language

We propose a simple, purely functional language for "implementing" L. We already mentioned the requirement for referential transparency in a candidate language for this role: it is absolutely crucial that L is a function in mathematical sense. Any impurity

in the language used for describing this mapping could compromise this property, even if unintentionally.[45]

The evaluation order of SILK is (at least currently) left unspecified: in writing specifications, it is best to assume strict evaluation, even though the actual implementation might be lazy. Because SILK is a Turing-complete language, it is clearly possible to write non-terminating programs – however, it is also fairly easy to ensure termination. Most specifications will only use structural recursion (whether directly or by using predefined combinators). Since the input data structures are always finite (stacks and heaps of the program being examined), structural recursion will always terminate, regardless of evaluation order. However, in cases where the programmer employs general recursion, they need to make sure that the specification will terminate under strict evaluation.

SILK is statically typed, with a simple predicate-based type system. It provides numeric types (initially only integers, although floating-point numbers may be considered as a future addition), function types, explicit scopes and algebraic data. In the context of LLVM property specification, it is extended with a few "builtin" types, namely for stack frames (activation records), stacks, heap objects and the heap. All these builtin types behave like explicit scopes.

The language itself is not a central topic of this thesis, and as such, we won't give an explicit or rigorous specification of the language. Instead, we will give a tutorial-style introduction and a few examples of use.

The most basic building block of a functional program is a function. Like in many other languages, SILK provides unnamed functions (lambdas) with named parameters, like this:

```
f = |x| x + 2
```

where `|x|` creates a formal parameter and binds the name `x` within the scope of the lambda. Functions in SILK are curried, i.e. the usual way to write multi-parameter functions is `|x| |y| ...`: a function of two parameters is written as a function of a single parameter whose value is a function of the other parameter. Type signatures can be given for values in SILK:

```
f : int → int
```

this declares `f` to be a function from integers to integers (and `→` is the type constructor of a function type). Function application is written as juxtaposition, i.e. `f 3` applies

[45] While not identical, we have experienced a closely related problem with CESMI: the implementation language in this case is usually either C or C++, providing a single pure value, the set of initial states, and a single pure function, the successor function. In practice, it often happens – due to entirely unintentional side effects in the implementation of one of those – that the reachable state space differs between invocations of the model checker.

function `f` to the value 3. Blocks of bindings are declared using the `def` keyword (and are, by default, recursive):

```
b = def
  g = f ∘ f
  f = |x| x * 2
end
```

Case analysis can be done using the `case ... of` construct, supplied by a number of lambda forms, where the parameters of the lambdas are patterns; bare words enclosed in |-brackets introduce new bindings, quoted names (using a single leading apostrophe) refer to existing bindings. If `cons` and `nil` are list constructors, we could write `map` as follows (where `case _ of` is a shorthand for `|x| case x of`):

```
map = |f| case _ of
  |'cons x xs| cons (f x) (map f xs)
  |'nil      | nil
end
```

When dealing with explicit scopes (recall b above), but also with other types of collections, we use full stop to "open" the scope or collection, like with C and C++ aggregate access syntax:

```
f = |array| (b.f ∘ b.g) array.3
```

Since bare words on the right of a full stop are interpreted as field names (i.e. they are quoted by default, as if preceded by a `'`), indexing collections using variables needs anti-quotation. While in the context of patterns, we believe a leading underscore is a good anti-quotation syntax, this does not fit so well with non-pattern uses. Instead, we treat comma specially when it is the first letter of a word. Finally, as a syntactic shortcut, we allow collapsing full stop and a comma into a single comma, giving us the following syntax:

```
index = |array| |idx| array,idx
```

which is equivalent to saying `(array . ,idx)`.[46] Also, since literal numbers are their own quotations (and anti-quotations), it is just as legal to write `array,3` (or `array,3,5` for multi-dimensional objects) as it is to write `array.3`.

[46] In this regard, besides parenthesis-like characters, `.` and `,` are quite special, since they are, unlike other operator characters, not allowed to occur in identifier names. In other words, full stop and a comma are interpreted as punctuation even when not surrounded by whitespace. To the contrary, when a comma is followed by a whitespace character, it is interpreted not as anti-quotation but as a delimiter in list syntax.

5.3.2 Source-Level Variables

One of the main challenges to overcome is that the relationship between various registers, memory locations and variable names as appearing in the source code is not always simple, or obvious at first sight. This is a problem that all source-level debuggers need to address, and there is a number of schemes for tracking the relationship in existence. Most such schemes are platform- and sometimes even architecture-specific. Fortunately, LLVM provides its own portable mechanism for tracking these relationships, in form of metadata blocks embedded in LLVM bitcode files (see also **Section 4.1.6**). These metadata blocks are readily available via standard LLVM APIs, and can be used to extract a map of variables provided by the compiler – in this respect, we have to rely on correctness of the compiler to obtain sensible data.
While it would be much easier and theoretically more robust to use low-level values in property specification, it would be of little practical use, as there is simply no way to predict how and where exactly will the compiler store variables at various points in time. Even if the programmer could determine which memory locations or which register values are of interest, this would be very compiler-specific, hard to write and even harder to comprehend.
An additional challenge arises from optimisation passes which may entirely eliminate source-level variables from the program. While this is not technically a problem for specification, it could easily lead to unexpected results: properties of the form "in all frames where variable x is defined, its value is greater than 5" will trivially hold if the variable x is eliminated and hence is not defined in any frame at all. Since pattern matching in SILK (at least for the purposes of defining atomic propositions) is required to be total, it is practically enforced that all variable access is guarded.

5.3.3 Traversing Stacks

The primary concern in writing expressions for use as atomic propositions is traversing stacks: for each thread (or a particular thread) look at all (or particular) activation frames. The top and bottom frames are often of special interest: the former tells us which function is currently executing and the latter which function is the entry point of the thread (this may not be the actual bottom-most frame though, as the runtime often owns the entry point, both for the entire program and for each thread as well). Among threads, the main thread could be of special interest – this is the thread with index 0.
Now consider the following SILK fragment:

```
main_running = |s| case s.threads,0 of
  |'nothing| false
  |'just _ | true
end
```

this evaluates to true whenever the main thread exists. Likewise, we could say

```
any_running = |s| not s.threads.empty
```

– this predicate is true when any threads at all exist – and demand that $\mathcal{G}$(any_running ⇒ main_running). This would express the property that when the main thread terminates, all other threads have terminated as well.
Now consider we had a function in our program (the program which we are verifying), say `lock_resource` with a local (implementation) variable called `waiting`. We could write this SILK expression:

```
waiting = |s| let
  in_lock = s.threads.filter (|t| t.top.function ≡ 'lock_resource)
in
  in_lock.any (|t| t.top.'waiting ≡ integer 1)
```

Here we first get a list of all threads for which the topmost frame belongs to the function named `lock_resource` and then check whether in any such threads, the value of the variable `waiting` in that topmost frame is 1. Now we could write an LTL formula that said $\mathcal{G}$(waiting ⇒ $\mathcal{F}\neg$waiting) – meaning that all waits are eventually resolved.
As an aside: since SILK is statically typed, the result type of keyword-based indexing (`top.'waiting` in this case) must be uniform across all keywords, for a particular object (`x.top` in this case). Since the type of the LLVM value is not known statically in the SILK program, runtime values are represented as an algebraic data type with a number of constructors:

```
value = type
  data integer : int → value
  data pointer : pointer? _ → value
  data struct  : struct? _ → value
  data none    : value
end
```

where `pointer?` and `struct?` are type predicates: the former declares that the value can be dereferenced to obtain a `value`, while the latter holds for types which can be keyword-indexed to yield a `value`.

5.3.4 Traversing the Heap

While traversing stacks is fairly easy – they constitute a collection of frames, which are in turn simple aggregate values (with local variables as constituents), heap is more tricky. While the heap is a collection of objects, these objects have no meaningful inner structure. The only structural information that DIVINE can provide about any heap object is the list of embedded pointers and the objects they point to. The types of the pointers are however not available. The only way to access heap objects in a manner

that reveals their inner structure is through typed pointers, which exist in stack frames as local variables (or if such pointers exist as global variables).
Typed heap objects, like local variables, are represented as a `value` in the SILK program, and can be obtained by dereferencing pointers that arise as local variables. Consider the following snippet, a variation on the `waiting` predicate from above, but this time the state of the resource lock is a heap-allocated C structure:

```
waiting = |s| let
  in_lock = s.threads.filter (|t| x.top.function ≡ 'lock_resource)
in in_lock.any $ |t| case t.top.'lock of
    |'pointer l₁| case l₁.deref of
        |'struct l₂| l₂.'waiting ≡ integer 1
        |_         | false
    |_           | false
```

Sometimes, however, untyped access to the heap is desirable: as outlined above, the only information DIVINE has – besides the size of the object and the actual raw bits stored there – is the list of pointers that point into the heap. The program state also contains a list of all heap objects: one obvious property we could express is that all allocated heap objects are reachable: this is the same as what DIVINE already provides as a built-in safety property, described in **Section 5.1.4**.

6 Reductions

As outlined in **Chapter 2**, state space reductions are a crucial building block of any successful explicit-state model checker. Such reductions exploit high-level properties of state space graphs, where a property can be shown to hold on the entire state space iff it holds in some particular subset of the entire space, and if such a subset of the state space can be computed efficiently. A successful reduction will save memory (required for storing the visited and open sets in the model checking algorithm), while also shortening the time required to explore the graph.

6.1 Partial Order Reduction

As introduced in **Section 2.7.1**, Partial Order Reduction exploits symmetries arising from interleaving of multiple execution threads, or multiple processes. Often, a local portion of the state space exhibits commutative behaviour: multiple interleavings first diverge due to partial application of effects, and immediately converge into a single state, as the effects of multiple independent action sequences are completed. In those cases, as long as the partial effects cannot affect the validity of the property under scrutiny, an entire region of the state space can be represented by one arbitrary path through this region. One such scenario is shown in **Figure 6.1**.

6.1.1 Background

The traditional dynamic partial order reduction is based on an efficient heuristic approximation of the full (theoretical) reduction. In this section, we briefly present the theoretical concept and the usual heuristics.

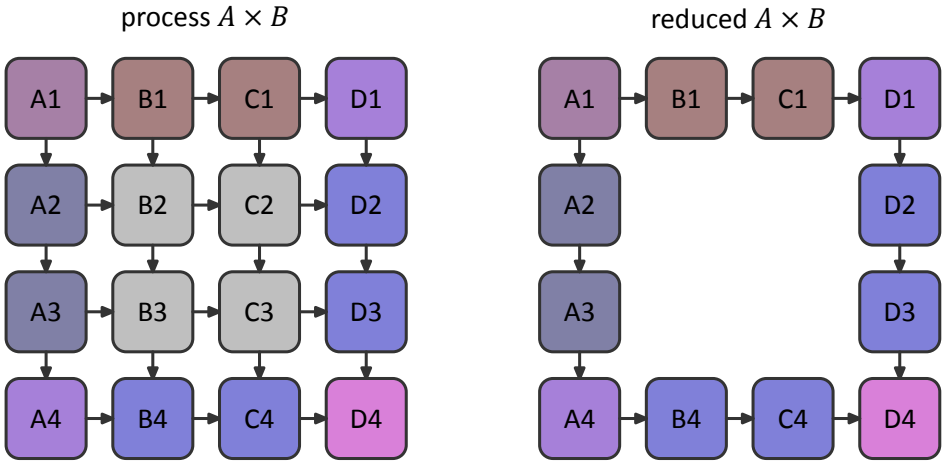


fig. 6.1 An example of POR – all the uncolored vertices and their edges may be left out of the explored state space, assuming that the transitions are all invisible and independent.

First, we shall give a definition of a Kripke structure [104] (as briefly introduced in **Section 2.1**), extended with atomic propositions:

def. 6.1 A Kripke structure is a tuple (S, T, S_0, L) where

- S is a set of states,
- T is a set of transitions ($\forall \alpha \in T : \alpha \subseteq SxS$),
- $S_0 \in S$ is an initial state,
- $L : S \to 2^{\mathcal{AP}}$ is a labelling function, with $\mathcal{AP}$ being a set of atomic propositions.

■

For simplicity, we will only consider deterministic finite systems. Each $\alpha \in T$ can therefore be seen as a partial function $\alpha : S \to S$. Practically, the extension to non-deterministic systems does not affect the results of this thesis.

def. 6.2 A transition α is *enabled* in a state s, whenever $\alpha(s)$ is defined. ■

The idea of partial order reduction is to disable some transitions in some of the states, obtaining a new structure K', such that for a fixed $\text{LTL}_{-\text{x}}$[47] formula φ, it holds that $K \vDash \varphi \Leftrightarrow K' \vDash \varphi$. The reduced system K' is defined through so-called *ample sets*. For each state $s \in K$, we define $ample(s) \subseteq enabled(s)$ to be the set of transitions enabled in the reduced system.
Apart from requiring correctness (the system defined through those ample sets satisfies φ iff the original system does), two properties are crucial for successful application of the reduction:

1. The ample sets need to be efficiently obtainable from description of the original system.
2. The reduction achieved needs to be significant, that is, the reduced system should be significantly smaller than the original.

In order to formulate a good approximation of the partial order reduction, we need to define two useful notions. First, we define a *dependency* relationship between two transitions:

def. 6.3 Given α, β transitions, we say that α is independent of β iff:

1. (non-disabling): $\forall s : \alpha \in enabled(s) \Rightarrow \alpha \in enabled(\beta(s))$
2. (commutativity): $\forall s : \alpha(\beta(s)) = \beta(\alpha(s))$

■

[47] By $\text{LTL}_{-\text{x}}$, we mean Linear Temporal Logic without the X (next) operator. Please refer to **Section 2.8** [50] for definitions of LTL and $\text{LTL}_{-\text{x}}$.

Finally, we need to be able to distinguish transitions which can be observed by the property (i.e. they change the set of atomic propositions that hold) and those that are *invisible*:

def. 6.4 We say that transition α is *invisible* with respect to $\mathcal{AP}'$ iff it holds that $\forall s \in S : L(s) \cap \mathcal{AP}' = L(\alpha(s)) \cap \mathcal{AP}'$. ■

When we refer to invisibility later in this thesis, we always refer to invisibility with respect to the alphabet of some formula φ.

6.1.2 Approximating POR

Traditionally, these four conditions are used to determine a suitable ample set for each state s:

C0 $ample(s) = \emptyset \Leftrightarrow enabled(s) = \emptyset$

C1 Along every path in the original structure K that starts in s, the following condition holds: a transition that is dependent on a transition in $ample(s)$ cannot be executed without a transition in $ample(s)$ occurring first.

C2 If s is not fully expanded, then every $\alpha \in ample(s)$ is *invisible*.

C3 (cycle proviso) A cycle in reduced structure is not allowed if it contains a state in which some transition is *enabled*, but is never included in $ample(s)$ for *any* state s on the cycle.

The conditions **C0** through **C3** are sufficient to guarantee obtaining correct ample sets. (For a proof, please refer to [50]). Conditions **C0** and **C2** are easily checked locally, and therefore their execution can be left intact for purposes of a parallel implementation of partial order reduction. A procedure for checking **C1** that is independent of search order is also available, such that whenever the procedure returns *true* for a given set of transitions, it is guaranteed to satisfy **C1**.

Additionally, we need a search-order-independent implementation of the **C3** check, since parallel model checking algorithms usually cannot guarantee DFS exploration order which is required by the usual heuristic. A parallel implementation of **C3** has been first proposed in [14]. Assuming **C1** holds for all ample sets along a cycle in the reduced structure, **C3** holds for this cycle whenever at least one state s on the cycle is fully expanded (meaning $ample(s) = enabled(s)$).

Usually some variation of this approximation (a proof that it is sound is again available in [50]) is used to implement **C3** checking in practice, using a depth-first search stack. Whenever a state is encountered that would close a cycle, it is fully expanded. However, this implementation heavily relies on depth-first search. Therefore, we need to replace

this condition with a different one, that would not rely on presence of a depth-first search stack.

6.1.3 Covering Cycles in Parallel

In this section, we will present an algorithm[48] that guarantees that along every cycle in the reduced structure, there is at least one fully expanded state, effectively implementing the **C3** check as described above. The algorithm is based on a variation of *topological sort* that can be efficiently implemented in parallel – unlike the traditional check based on DFS, the so-called "in-stack" check [50]. The more traditional DFS-based proviso is used in DIVINE in conjunction with its Nested DFS implementation.
In addition to checking **C3**, we require the **C3** check to avoid interfering with a desirable algorithm property called on-the-fly execution (cf. **Section 2.6.7**). This means that if the algorithm is able to produce a counterexample without exploring the full state space, checking **C3** should not prevent such an algorithm from doing so. We will discuss this property later on.

def. 6.5 A transition graph $G = (V, E)$ induced by a Kripke structure $K = (S, T, S_0, L)$ is a graph (V, E) such that $V = S$ and $(s, t) \in E \Leftrightarrow \exists \alpha \in T : \alpha(s) = t$. ■

We assume that the model checking algorithm is not concerned with the transitions of the Kripke structure and instead explores its induced transition graph.
We also assume that the model checking algorithm is based on accepting cycle detection and is invariant under exploration order. This means that the algorithm is correct independently of the order in which it explores new transitions, as long as it eventually explores each transition reachable in the reduced state space. Moreover, if the algorithm requires revisiting states, we assume that it is possible to defer these revisiting operations arbitrarily long in the execution of the algorithm. This is not crucial for correctness, but it is important for the algorithm to keep its asymptotic complexity under the proposed reduction algorithm.

---

alg. 6.1 The algorithm for checking **C3** is based on a procedure that finds a set of states that covers all cycles. From there, it is easy to build the reduced state space incrementally. The state of the algorithm is described by `s` the set of processed states and `full`, the set of edges that do not exist in `ample` but have to be added to the reduced state space to fullfill **C3**.

```
data C = C { _s :: [S], _full :: [E] }
```

The main entry point of the algorithm is the function `c3` which needs, in addition to the original problem instance, an alternative definition of `es`, i.e. the set of edges of the

[48] The C++ implementation of the cycle proviso algorithm can be found in the file `divine/graph/porcp.h` in DIVINE source distribution.

graph, called `ample` – this, as the name hints, only contains the edges that are included in the ample sets computed by applying **C0** through **C2**.

```
c3' :: M → [E] → Computation ([S], [E]) C
c3' m@(vs, es, as, is) ample done = forever $ do
  s₁    ← get s
  full₁ ← get full
  let s₂ = reachability (vs, ample ∪ full₁, as, s₁ ∪ is)
  (expand, _) ←  callCC $ cover_cycles (s₂ \\ s₁) ample []
  let full₂ = full₁ ∪ [ (u, v) | (u, v) ← es, u ∈ expand ]
  when (empty expand) $ done (s₂, ample ∪ full₂)
  full ⇐ full₂
  s    ⇐ s₂
```

The helper function to find a set of vertices such that every cycle in a subset of the state space contains at least one of the included state is then implemented as follows:

```
cover_cycles :: ∀s. [S] → [E] → [S] → Computation ([S], [E]) s
cover_cycles vs es pre done = do
    when (empty vs) $ done (pre, [])
    if (empty tail) then recurse (take 1 vs) (pre ∪ take 1 vs)
                    else recurse tail (pre ∪ pre_tail)
  where pred v = [ u | u ← vs, (u, v) ∈ es ]
        succ u = [ v | v ← vs, (u, v) ∈ es ]
        tail = [ t | t ← vs, empty $ pred t ]
        pre_tail = cat [ succ x | x ← tail ]
        recurse rm expand = cover_cycles (vs \\ rm) es expand done
```

The algorithm starts with both `s` and `full` empty:

```
c3 m ample = compute (c3' m ample) $ C [] []
```

---

stmt. 6.1 The function `cover_cycles` in **Algorithm 6.1**, given a set `vs` of states and a set `es` of edges, returns a set `pre` of states such that for every cycle $c \subseteq$ `vs`, it holds that $c \cap \text{pre} \neq \emptyset$.

proof The main recursion invariant is that a cycle $c \subseteq$ `s` is either fully embedded in `vs`, or there is a state $v \in c \cap$ pre. This is clearly true before entering the loop for the first time, as `vs` contains all states (vs = s). When the algorithm terminates, vs $= \emptyset$, therefore, for each cycle $c \subseteq$ s, it must hold that $c \cap \text{pre} \neq \emptyset$.
We now only need to show that this is indeed the loop's invariant. First, a state v is never removed from `vs` if it is a part of a cycle fully embedded in `vs`, because it can never appear in `tail`. This however means that such a state might only be removed when `tail` becomes empty, which also means that it is added to `pre` at the same time. This means that if v has been a part of a cycle fully embedded in `pre`, then $v \in c \cap$ pre.

Termination: The algorithm clearly terminates, as in each iteration, at least one state is removed from vs. □

stmt. 6.2 The time complexity of procedure `cover_cycles` in **Algorithm 6.1** is in $\mathcal{O}(|S| + |E \cap (S \times S)|)$.

proof Each state in S (vs) is examined exactly once, when it is being removed from vs. When a state is being removed, each of its outgoing edges pointing back into vs is examined exactly once. □

stmt. 6.3 **Algorithm 6.1** ensures that on every cycle in the reduced state space, there is at least one fully expanded state.

proof We show correctness of **Algorithm 6.1** by induction. As a base, let us consider that s_1 in `c3'` is empty. It follows from **Statement 6.1**, that result of the re-expansion is exactly the set of edges such that at least one state on every cycle is fully expanded.
We can now assume that before every invocation of re-expansion (the `reachability` call in `c3'`), s_1 already has, on each cycle, at least one fully expanded state (i.e. the corresponding edges have been added to `full`). We need to prove that after executing the following `cover_cycles`, s_2 in conjunction with `expand` will retain this property.
Again, from **Statement 6.1**, we can deduce that every cycle fully embedded in $s_2 - s_1$ will have at least one fully expanded state (that is, there will be at least one state on every cycle, such that all its enabled edges will be explored).
This covers all cycles that do not cross the s_2 / s_1 boundary. However, when there is an edge (v, w) such that $v \in s_1$ and $w \in s_2 - s_1$, we know that v has been fully expanded. Clearly, any cycle crossing the s_2 / s_1 boundary will contain at least one such edge.
When **Algorithm 6.1** terminates, s contains all of the reduced state space. □

6.1.4 Time Complexity

Clearly, it is important that a prospective **C3** check can be performed in *linear* serial time – an algorithm with super-linear complexity would clearly impede the performance of the process of state space exploration, and in turn of model-checking.

stmt. 6.4 Time complexity of **Algorithm 6.1** is linear in size of the reduced state space.

proof Every invocation of function `cover_cycles` in **Algorithm 6.1** is linear in its parameter vs (**statement 6.2**). We show that any given state is in vs in at most one invocation of `cover_cycles`.
When a state occurs in vs, it will be immediately added to s. When a state is already in s, it will never again occur in vs. Therefore, we conclude that every state occurs in vs in at most one iteration. □

Even though the **C3** check itself is linear, many of the parallel accepting cycle detection algorithms are not linear in all cases. There are two requirements for the combined algorithm for accepting cycle detection in the reduced state space: firstly, the combined algorithm should have time complexity no worse than the original accepting cycle detection algorithm employed and secondly, it should not be required for the **C3** check to perform a full reachability pass over the state space before starting the cycle detection itself.

stmt. 6.5 Using **Algorithm 6.1** does not affect time complexity of the underlying model checking algorithm.

proof Any algorithm for accepting cycle detection needs $\Omega(|V| + |E|)$ time. Since we require the model checking algorithm to allow deferring any revisiting operations, we can assume that the algorithm can be reordered to run in two passes: first, it explores the state space in $\Theta(|V| + |E|)$ and then it possibly carries out additional computation with complexity t. Clearly, **Algorithm 6.1** only interferes with first of these two passes, and since it runs in $\mathcal{O}(|V| + |E|)$, the overall complexity of the first pass is $\Theta(|V| + |E|)$, making the overall complexity $\Theta(|V| + |E|) + t$, which is the same as that of the original model checking algorithm. □

Further, we shall consider the proposed heuristic in terms of on-the-flyness of the underlying accepting cycle detection algorithm. It can be seen easily, that for a level 2 on-the-fly algorithm (see **Section 2.6.7**) with the required properties, the algorithm would clearly stay a level 2 on-the-fly algorithm when combined with the heuristic. Unfortunately, no such algorithm is currently known.
As for level 1 algorithms, these are not required to terminate early for every input, even in the cases where there is a counterexample to be found in the state space. Moreover, since the algorithm is invariant under exploration order, the ability to find a counterexample is largely dependent upon the order in which the state space is explored.
Since level 1 on-the-flyness is of a relatively heuristic nature, the following is not really a theorem – strictly speaking, to prove this property, it would be enough to find an input where the algorithm finds such a counterexample, and where $ample(s) = enabled(s)$ for each state s. However, this hardly tells us anything about practical behaviour of the reduction.

stmt. 6.6 A model checking algorithm that is level 1 on-the-fly will also be level 1 on-the-fly if combined with **Algorithm 6.1**, while maintaining this property to an useful degree.

stmt. 6.7 If the original algorithm would find a counterexample in a small fraction of the state space, a relatively small change in exploration order induced by the reduction algorithm is unlikely to affect size of this fraction significantly.
Moreover, if the probability of discovering a counterexample in a given percentage of state space is independent of the exploration order, then **Algorithm 6.1** will not alter this probability at all.

6.1.5 OWCTY with POR

To successfully exploit the partial order reduction algorithm presented, it needs to be combined with a suitable model checking algorithm. In this section, we show that OWCTY [42 and 72] fulfils all the restrictions we have placed on the model checking algorithm. Moreover, it can be adapted to run on-the-fly and it is generally suitable for practical model checking [11].
The algorithm starts out with a single full reachability (and a heuristic may enable it to uncover a counterexample during this phase, making it on-the-fly). This reachability pass is exploration-order-independent. We incorporate the **C3** check proposed in previous chapters into this pass. This also means, that no re-visits are done before the **C3** check is complete.

alg. 6.2

From a high-level point of view, the modified OWCTY works by tweaking the instance of the model checking problem before passing it along to regular OWCTY – all it takes is to restrict the vertex and edge sets based on the results of the `c3` procedure (cf. **Algorithm 6.1**).
In a practical implementation, the **C3** check would be integrated more tightly into the initial reachability (initalisation) phase of OWCTY (which also sets up bookkeeping for a more efficient implementation of the OWCTY algorithm itself).

```
owcty_c3 :: M → [E] → [S]
owcty_c3 m@(_, es, as, is) ample = owcty m'
    where m' = (vs', es', as, is)
          (vs', es') = c3 m ample
```

For details about the reachability and elimination procedures, please refer to **Section 2.6.5**. The property important here is that they re-explore the state space already stored in `s`. Moreover, it is not necessary to store edges explicitly – we only need a single bit for each state, remembering if *enabled*(s) or *ample*(s) has been used for this state. The successors are generated from the description of the input and description of the state being expanded.

6.2 $\tau-$ and $\tau+$reduction

Both reductions presented in this section are instances of a combined partial-order/path reduction as introduced in **Section 2.7.1**. Both are approximations based on specific properties of LLVM state spaces. A more traditional POR implementation is difficult, because there is no global knowledge about system transitions and specifically their dependencies. The main problem is the availability of unrestricted pointers: globally, data dependencies in an LLVM program are determined by pointer values possible at runtime. While a global analysis of dependencies in a program seems conceivable, it

also appears to be a very complex undertaking. While it might offer improvements over the much simpler approximations described in this section, we do not expect the benefit to be large enough to offset the large investment in research and development and the high expected cost of the required upfront analysis.

6.2.1 τ-reduction

This reduction is based on the observation that some transitions in the state space are invisible for other active threads in the system. We shall call these actions τ; such actions of a thread or a process can be delayed over other τ actions of other processes. In fact, multiple subsequent τ actions of a single process/thread can be safely collapsed into a single transition without any effect on other threads or the observed property. In traditional model-based model checking of asynchronous systems, this reduction would be quite ineffective, because τ chains are fairly rare in purpose-built models. On the other hand, they are extremely ubiquitous in LLVM bitcode. While identifying all τ actions could be very complex, there is a very simple yet very efficient heuristic that can identify and collapse a majority of safe (i.e. not forming a loop) τ transitions. All transitions that:

- do not access memory (they can still read and write registers), and
- are within a single basic block

can be safely treated as τ transitions. From experience, we know that assembly-level programs, especially in the RISC style with explicit loads and stores (as is the case of LLVM), contain a significant share of instructions (actions) that meet both these criteria. This reduction is closely related to "superstep POR" proposed in [158], although simpler. One way to improve both these reductions is an algorithm that identifies memory writes that are invisible to other threads (such an algorithm is discussed in more detail below, in **Section 6.2.3**). On the other hand, since the LLVM virtual machine has a possibly infinite register file, no register spilling happens (register spilling is normally a major source of invisible memory writes; in LLVM-based compilers, register allocation takes place in a later phase of code generation). This naturally limits the number of invisible writes, and in part explains why the reduction is so successful despite its simplicity.

6.2.2 τ+reduction

A simple way to approximate both partial order reduction and path compression is to keep a single thread running as long as cycle and observability criteria are met. In the instance of τ-reduction described above, the observability criterion states that an instruction is observable iff it affects the content of shared memory: this approach is inherited without change by τ+reduction. The difference lies in the cycle check. In τ-reduction, any branching (jumping) instruction is treated as possibly closing a control flow cycle (since branching instructions always cross basic block boundaries), forcing

an intermediate state to be generated. However, if we defer the cycle check, we can do much better. Especially in optimised code, branching easily dominates memory access, and the static proviso becomes a major source of inefficiency. In lieu of a simple static check for a branching instruction, we can dynamically detect control-flow loops at successor generation time.

The control location of a thread is kept using a "program counter", a 4-byte integer value that uniquely identifies a specific instruction. Clearly, any actual loop in the program will traverse a single control location twice – hence, it will also encounter the same program counter value. With this in mind, we keep a set of program counter values that we traversed while looking for a successor. Only when an actual control flow loop closes, we interrupt the execution and generate a new state. Each time a successor is generated, the visited set is cleared.

While this is still an approximation, since the (unobservable) loop may finish in finite number of iterations, it is very cheap to compute. Keeping track of full system configurations – an approach that would achieve a better reduction for data-dependent loops with no memory access – would be much more computationally expensive. We reckon that tracking the comparably minuscule program counter value is a viable compromise.

From the model checking perspective, $\tau+$reduction deals with successor states and state spaces, replacing diamonds and chains with one-step transitions. This view is useful for arguing correctness and when thinking in terms of systematic exploration, and corresponds more closely to the language of ample sets as defined for traditional POR (cf. **Section 6.1**). However, from the point of view of a single execution trace or from the point of view of the program being executed, this view is less appropriate. Therefore, we formulate an alternative, equivalent view of the reduction in terms of interleaving (also called interruption) points.

We define an interleaving point as a place "in-between" two instructions in the program text, where a context-switch (rescheduling) of threads (from the point of view of the program) might happen. When building an unreduced state space, an interleaving point is inserted between each pair of instructions. This intuitively captures what happens in a real CPU, whether a single core time-sharing multiple threads, or an actual multi-core unit. However, as outlined above, not all interleavings cause observable differences in behaviour of the program. τ-reductions then act by removing some of these interleaving points. τ-reduction simply inserts an interleaving point right before each `store` and each branching instruction, statically.

On the other hand, $\tau+$reduction, as a semi-dynamic technique, acts on the program as it is being executed. First, interleaving points are inserted before all `store` instructions, just as with τ-reduction. Then, more are created and removed on the fly: whenever a thread closes a control flow loop, an interleaving point is inserted just before the first instruction that would have been repeated. After the re-scheduling happens, this interleaving point is then dropped again, since a non-looping execution might pass through it at other times. Apart from technical requirement of the model checker that each step is finite, these loop-related interleaving points are intuitively required to avoid delaying other threads indefinitely.

6.2.3 Store Visibility

In **Section 4.4.2**, we have outlined that with exact pointer tracking, it is possible to decide whether a given area of memory is visible to any given thread. Likewise, it is possible to decide whether a heap object is private to a particular thread, i.e. for all other threads it falls outside their area of visibility.
Since writes to such "private" heap objects cannot be observed by other threads, we can mark the corresponding `store` instructions as unobservable for the purposes of $\tau+$reduction, again substantially improving its already very good efficiency.
In order to effectively identify the relevant `store` instructions, we trace the root set excluding the currently executing thread. If the heap object that is being written to is not encountered in this manner, then the write is invisible, since no other thread can read the corresponding memory location. Since we use tracing, this remains true after any combination of `loads` or pointer manipulation. The only action that would make the `store` observable would be a different `store` *in the same thread*, writing a pointer to the relevant object into a pre-existing, already shared memory location. However, since this must happen in the same thread, the change caused by first in such a sequence of `stores` can never be observed, and the later `store` will properly cause an interruption point to be inserted.

6.2.4 Evaluation of Efficiency

Informally, model checking of multi-threaded C and C++ programs without any sort of reduction is nearly impossible. On the other hand, with $\tau+$ reduction, we have successfully verified a number of non-trivial models. To put this into a more formal perspective, we have compiled a table of state space sizes without reduction (only models of a size that can be reasonably checked without reductions are included here; in other words, the models used in this section are comparatively very small). The "full" reduction is $\tau+$ with memory access visibility enabled (the other reductions disregard memory visibility).

model	unreduced	τ	relative	$\tau+$	relative	full	relative
anderson	298 335	32 121	11.0 %	19 482	6.5 %	572	0.19 %
peterson	603 196	77 397	12.8 %	51 807	8.6 %	8 318	1.38 %
ring	4 625 684	461 663	10.0 %	251 599	5.4 %	15 368	0.33 %
szymanski	9 502 590	1 220 399	12.8 %	807 521	8.5 %	4 967	0.05 %
lamport	527 886 892	23 915 110	4.5 %	12 615 138	2.3 %	485 563	0.09 %
global	1 437	528	36.7 %	251	17.5 %	62	4.32 %

From the table, we can see that in general, bigger models exhibit greater savings – in line with our expectations. Moreover, we can see that each level of reduction yields substantial returns, justifying the default mode where all reductions are enabled. Finally, it is entirely possible to verify models with many millions of states after full reduction

– with a little extrapolation, we can guess that their full state space would be on the order of billions of states.

6.3 Heap Symmetry Reduction

In **Section 4.4**, we have introduced the concept of a program heap, or dynamic memory, treating it like an oriented graph, with objects becoming vertices and pointers in those objects becoming edges. With regards to program behaviour, the exact layout of a heap in memory is usually irrelevant (bar pointer manipulation or indexing bugs) – however, it affects the actual bit-level representation of a program state.
This introduces a degree of symmetry into the state space of the program, where multiple distinct states may only differ in heap layout. Since the behaviour of the program is not affected by this difference, we obtain multiple mirror copies of a subset of the state space. This can be extremely wasteful, and is most pronounced when multiple threads are using the heap (which is a common case). Whenever allocations can become interleaved, two symmetric successor states arise, differing only in the ordering of the two heap objects in the physical address space. It is very desirable to detect and exploit this symmetry to reduce the state space.
There are two main ways to implement symmetry reduction. One is based on a modified state comparison function, which detects symmetric situations and makes any two symmetric states equal. The major downside of this approach is that it precludes use of hash tables – the structure of choice in explicit-state model checking. The other option is canonisation: a technique where each state is transformed to obtain a canonic representative of each symmetry class. This way, all symmetric states are represented by the same bit vector, and standard equality and hashing can be used.
On the flip side, detecting symmetric heap configurations is much easier than constructing a canonic representative. This is especially true for programs with explicit (manual) memory management. In some programming languages[49], the heap is subject to automatic garbage collection, and while LLVM has optional garbage collection support, it is not used when compiling C or C++ programs. If exact collection is used [101], all pointers must be tracked by the runtime, especially if using a copying (or more generally, moving) collector. If this information is available, it can be used to implement heap canonisation. In fact, a slightly modified single-generation copying garbage collector will produce a canonic heap layout after every collection cycle.
Opposite to languages with automatic memory management, languages like C and C++ require memory to be explicitly `free`-d to allow memory re-use and avoid resource leaks. However, this also means that the C runtime puts very little constraint on how pointers can be manipulated, since correct memory management is the responsibility of the program, not the system. Unfortunately, this makes it impossible to retrofit garbage collection (and analogically, heap canonisation) to these languages while retaining

[49] Or, more exactly, programs, since garbage collection can be implemented for specific programs even in languages without intrinsic garbage collection support.

full generality. In theory, it is legal for a C program to save pointers to a file and read them back later for further use, or to store them bit-flipped in memory or even `xor`'d together as in a `xor`-linked list. Such obscured pointers are however extremely rare in actual programs, and we can make them illegal for the purpose of verification. Basically, addition is the only reasonable operation to do on an (integer-casted) pointer value; an error can be raised when attempting any other manipulation. In most circumstances, a non-additive operation on a pointer would indicate a bug in the program.
Finally, in a controlled environment (i.e. when each instruction can be freely instrumented), obscured pointers are the only major obstacle in implementing heap canonisation. Therefore, restricting those, it becomes possible to fully track heap pointers throughout the program, as we have discussed in detail in **Section 4.4.1**, and based on this information, compute a canonic heap representation, adjusting all pointers accordingly. The actual layout we chose is based on DFS pre-order, with root pointers forming the initial search stack, global variables first, then deepest frame of the first thread and traversing stacks upwards first, then threads from the lowest thread-id to the highest.

7 Abstraction & Refinement

As discussed in **Section 2.1.1** and **Section 2.4**, data processed by a program pose a significant challenge to explicit-state model checking. In theory, it is perfectly valid to replace program inputs with explicit non-deterministic choice (in our case, using the `__divine_choice` builtin with a suitable parameter corresponding to the size of the data type in question). However, while this approach is sound and simple, for non-trivial inputs it becomes quickly intractable in practical terms.

Hence, some sort of symbolic approach is required to overcome this limitation, making analysis of more-or-less open systems feasible. The symbolic method that is most readily combined with explicit-state exploration of a state space is automated abstraction and refinement. The most commonly employed abstractions are based on predicates, and most common refinement techniques are based on (spurious) counterexample analysis. The latter method is generally known as CEGAR, or Counter-Example-Guided Abstraction Refinement.

While traditional abstraction engines work as interpreters, abstractions can also be "compiled" into programs. Instead of (re-)interpreting instructions symbolically, the abstract, symbolic instructions can be translated into equivalent explicit code which computes with symbolic values. For example, in the case of predicate abstraction, the resulting bitcode can directly manipulate and use predicate valuations instead of concrete variables, encoded as bit vectors. An abstracted program after such a transformation is basically a "normal" computer program, with somewhat unusual properties, but otherwise very similar to any other concrete program. The main exception is that the abstracted bitcode needs to make non-deterministic choices – the abstraction will have removed some information from the program, and as such, lose precision. This loss of precision directly translates into loss of determinism in program behaviour.

7.1 LART

LART is an acronym, standing for LLVM Abstraction and Refinement Tool. It is a work in progress and is primarily a vehicle for implementing abstractions (and their subsequent refinements) as bitcode transformations. Eventually, the goal of this tool is to make implementation of new abstract domains and new abstraction heuristics both easy and re-usable. By providing common scaffolding – pointer analysis, variable substitution, constraint propagation and so on, and also a number of "worked examples" of abstract domains, we expect that implementing new abstract domains for experimental evaluations becomes much easier than it is now. Second, by producing standard LLVM bitcode as its output, the abstracted programs can be readily analysed using pre-existing tools, including DIVINE and existing symbolic debuggers (possibly by compiling the abstracted bitcode into a native binary).

Especially the ability to produce runnable binaries makes this approach attractive – by substituting a suitable function for non-deterministic choice – whether interactive

asking the user for directions, prepared ahead of time in the form of "test vectors" or even randomised. Other than non-determinism though, the LLVM bitcode produced by LART is an ordinary program. Even runtime analysis tools like `valgrind` could be reasonably applied to such program – this might be useful on its own, for cutting away irrelevant details of the program (the same way it is ordinarily used with model checking), or as a support method for further analysis of counterexamples generated by a model checker.

7.2 Predicate Abstraction

One of the most successful abstraction techniques is predicate abstraction, where a number of predicates is established, describing the state of the program in some "useful" way. This could, for example, describe qualities of single variables, or relationships between multiple variables, like $a > 5$ or $a = b$. The program state is then encoded, instead as a valuation of variables and memory locations, as the set of predicates that are satisfied for a given state. If we understand the program as a state transformer, where the state is represented as a valuation of variables, we can derive an abstract program which instead transforms predicate valuations. As an example, take a program of the form $x \leftarrow x + 1$ and a predicate $x < 5$. If we represent the abstract program as a state transition system, we obtain two states: $x < 5$ and $x \geq 5$, and under $x \leftarrow x + 1$, starting in $x < 5$ we can either stay in $x < 5$ or move to $x \geq 5$, while $x \geq 5$ is invariant under this particular program fragment (assuming infinite integer arithmetic). What we see is that the program is no longer deterministic: since the exact value of x is not known, we don't know whether incrementing it will cross the boundary, 5, or not – we assume that either could happen.

This is clearly a loss of precision, as we would expect – the state of the program is encoded in a single bit, even though the real state space of the concrete program is infinite. Even though most of the information about the program is lost, it is still possible to prove interesting properties – particularly, formulas that can be expressed in terms of the available predicates can be, in some cases, shown to be true: as long as the loss of precision is not too big. For initial states where $x \geq 5$, for example, we can easily prove $\mathcal{G}(x \geq 5)$ for a program that increments x in a loop (or, with some minor additional assumptions, $\mathcal{FG}(x \geq 5)$ for an arbitrary initial state).

Consequently, due to the loss of precision, it is very likely that "unreal" paths will exist in the abstract program – those that do not correspond to any actual execution of the concrete program. As long as all such "unreal" paths satisfy our desirable property, this is not a problem: we only use abstractions that are "over" with respect to the properties we are interested in. For properties universally quantified over all executions (the case of all LTL properties), any approximation that does not remove executions will be "over", as long as it preserves truth of path formulae exactly (see also **Section 2.4.3**). With those provisions, there are four cases we need to understand with regards to path formula outcomes vs. path "realness".

- $p \not\models \varphi$, p is real: the property is violated by this run
- $p \not\models \varphi$, p is unreal: the abstraction is too coarse
- $p \models \varphi$ does not influence the outcome, whether p is real or not

Predicate abstraction, or in fact any abstraction, can be either *relational* or *non-relational* – the former type is more complex to work with and more precise, while the latter is a fair bit simpler to implement but also substantially less precise. The loss of precision arises from necessarily losing relationships between different values that have interacted. This is best illustrated by value assignments where the assigned value depends on a different abstracted variable, say $x \leftarrow y + 1$ – in this case, non-relational abstraction has no way of remembering the relationship of x and y, merely copying the representation of y into the value of x, suitably adjusted. Consequently, if x and y are later compared, the information cannot be used anymore: a relational abstraction might be able to tell that $x < y$ is true[50], but this would be often impossible for a non-relational abstraction.
Abstraction, and predicate abstraction in particular, is very simple in principle. Predicate valuations are easy to manipulate and the rules for doing so are usually easy to derive. This still leaves the question of which variables to abstract and which predicates to use to do so. Clearly, we could pick both these sets randomly – although doing so is unlikely to give very good results. A common strategy is to start with some *maximum* (or coarsest) abstraction – resulting in a program that exhibits all possible behaviours, maybe constrained by its control flow graph. Likely, many of the behaviours of such highly-abstracted program will violate the property of interest. This kickstarts the *refinement loop*, which is then responsible for picking suitable predicates (or alternatively, for finding other ways to refine the current abstraction). We will discuss the refinement step in more detail in **Section 7.4**.

7.3 Counterexample Analysis

The goal of counterexample analysis is to decide whether an abstract counterexample is real, i.e. that it corresponds to a counterexample in the original, concrete program. One way to do this is to substitute specific values for each indeterminate value (using the concretisation function γ) in a way that preserves all constraints on the abstract values, and simulate the counterexample in the concrete system.
This is in fact the only task where the abstraction engine and the model checker need to interact in a non-trivial fashion. For this reason, an interface needs to be established that will facilitate this exchange of information. The first step is to combine the abstract counterexample with the original program, which can be achieved by synthesising a new version of the concrete program that takes particular choices deduced from the abstract counterexample.

[50] We have been using classical arithmetic in all examples in this section. In practice, most programs use fixed-width bit vectors for representing integers, sometimes with special rules for over- and underflow. We will discuss this in later sections.

To this end, it must be possible to discern such choices from the counterexample, a process which has two prerequisites: the model checker needs to provide a counterexample in a machine-readable form, and the program must be able to annotate its steps with extra information, which is then included in these machine-readable counterexample traces. We have discussed both these aspects of counterexamples DIVINE generates for LLVM bitcode in **Section 4.7**.

7.4 Refinement

The process of refinement is easily the most complex part of a system based on automated abstraction. There are multiple reasons for this: first, each abstract domain has a different refinement strategy, second, refinement is largely a heuristic process. The mechanics of "refinement proper" are, however, quite simple: it is a matter of running the abstraction pass on the original (concrete) program again, with a new set of parameters[51]. It is the process of finding this new set of parameters that we will call "refinement" from now on.

With predicate abstraction, the method most commonly used for finding suitable new predicates is interpolation [117], based on an idea from logic. In this context, interpolation is the process of finding a formula ρ such that for some $\varphi \implies \psi$, the symbols in ρ appear in both φ and ψ, while $\varphi \implies \rho$ and $\rho \implies \psi$ hold. This formula can then be turned into a predicate and added to the working set, refining the abstraction (adding a predicate can never coarsen an abstraction, although it might fail to refine it). Using interpolation for computing the new predicate gives us an assurance that the refinement is "good" in some sense (i.e. it moves the refinement in a direction that will remove the spurious counterexample).

For value abstraction, a feasible refinement is based on adding complementary abstract domains to the representation of a value. We can imagine combining eg. parity and sign domains for a single variable, yielding the negative/odd, negative/even, positive/odd and positive/even abstract values. This can be achieved by simply computing multiple abstract domains for each concrete variable and for queries on those variables, give the most precise answer available from any of the domains – i.e. combine answers in a way that both *true* and *false* trump a *maybe*. I.e. a positive/odd integer x will yield, when computing the predicate $x \geq 0$, *true* and *maybe* for the respective abstract domains, yielding *true* in combination.

7.5 Implementation

As mentioned earlier, a good implementation strategy seems to be a LLVM-to-LLVM transformation, "compiling away" some details in the program, like exact values of

[51] Special considerations could make this process more complex – especially a desire to re-use data from a previous run of the model checker could introduce extra overhead. We defer the question of how to build an incremental system to future work.

particular variables. This way, we can reuse the abstraction engine in many existing tools.
The interesting questions that arise are: what kinds of abstractions to implement and how to refine them. A first and in a way the simplest candidate is to eliminate some variables entirely, as well as any values that depend on them, and replacing all branches they affect with non-deterministic choice. The crudest interesting refinement is then to put one of the variables back.
When variables associated with a particular loop are abstracted away in this way, the loop will have no effect and can be optimised away by a (combination of) pre-existing LLVM pass(es). If some but not all variables affected by a loop are abstracted away, the loop may stay but become significantly simpler. For choosing a variable to refine, a spurious counterexample could be quite helpful, as it may reveal which (abstracted) variables participated in the branching it took, although a crude refinement strategy could just pick variables at random (of course a decision must be made that a refinement is required, which means the abstraction/refinement driver decided there is *some* spurious counterexample, but it may not be available in a form suitable for the abstraction/refinement engine).
A more sophisticated abstraction technique is to replace concrete variables with a set of predicates. This allows for subtler abstractions and also subtler refinements. In case the abstraction fails, a counterexample is required to make reasonable refinement though. A refinement may either add more predicates, or possibly un-abstract a variable entirely (especially if it seems to already have suspiciously many predicates associated). In order to make better choices about refinements, the abstraction/refinement engine can annotate the LLVM bitcode with extra information, which would then show up in counterexamples. Of particular interest are annotations about path constraints on variables: if a spurious error occurs on a path guarded by *if (x > 1000)*, it's very useful for predicate refinement to see this in the counterexample. An intrinsic call, eg. `@llvm.dbg.assume` with a metadata node as a parameter, could serve that purpose fairly well. The job of the downstream tool is basically to include these "semantic metadata" instructions in their counterexample traces, so that the abstraction/refinement engine can pick them up and base its decisions on them. It's also up to the abstraction engine to add relevant `@llvm.dbg.assume` calls to the bitcode (additionally, the metadata nodes need to be able to refer to "live" values in LLVM registers, which then need to be passed as actual parameters to the assume; this is mostly an implementation detail).

7.5.1 Incremental Refinement

Transforming the LLVM bitcode obscures the relationships between various abstractions/refinements of the same concrete program. This is especially true if further transformation passes are applied after the abstraction. It would be possible for the refinement process in the abstraction engine to mark up the code regions it has changed compared to the abstraction it was refining (this would preclude use of standard LLVM transforms on the abstracted code though, as those would not be able to preserve this

kind of markup). Additionally, fairly elaborate support would be required on the side of the model checker to make use of such markup, and it's not clear if incremental verification would out-weigh the benefit of stacking standard LLVM transforms after the abstraction. After a non-incremental scheme is implemented, it might be worthwhile to measure how much work is being re-done by the model checker upon refinements, and possibly figure out a way to re-use it. This might be easier in a bounded model checker than in an explicit-state one.

note An option for explicit-state model checker might be to arrange the predicates in a bit-vector with "free" slots. While the predicate bitvector width does not change, the model checker can re-use the visited set and start exploring from a subset of already visited states, using the new program text. The abstraction/refinement component would need to keep the bit-to-bit memory layout of the program intact, and we would need to map program counters.

7.6 Alias Analysis

In simple cases, aliasing information is not necessary for abstraction. However, for partial abstractions (and especially refinement), alias analysis is vital for obtaining efficient transformations and fine-grained refinement control. Consider cases where abstract values are stored by the program in address-taken variables (i.e. in LLVM memory as opposed to LLVM registers). The addresses of those variables can be passed around and a code location not obviously def-use related to the origin of the value can load the value into a register. There are two options to address this issue: if the entire alias set the particular variable belongs to is abstracted together, loads can be statically and reliably marked up as abstract or concrete. When the alias data is too coarse though, and a finer abstraction control is required, some loads may become ambiguous.
To correctly transform ambiguous loads, an expensive runtime tracking mechanism is required to identify abstract values (and possibly their types when different variables use different abstract domains). This requires additional global storage proportional in size to the size of an alias set, and insertion of possibly complex branching at the point of each ambiguous load/store. The cost of implementing these transformations may be prohibitive: in this case, we may entirely drop support for "mixed" abstraction (with ambiguous loads), or we may instead produce bitcode with substantially altered semantics and expect the backend tool to implement runtime value tracking.

7.6.1 Abstract Memory

In a running program, each memory (heap) allocation will result in a concrete memory location being created in the program's address space. For the purpose of alias analysis, we need to abstract those concrete memory locations in some fashion, as it is impossible to statically compute the set of concrete locations. The common abstraction used for

this is assigning a single abstract location to every callsite of a memory allocation routine.
In other words, one abstract location is created for each static call that allocates memory; however, this is not the only option. A coarser approach could only distinguish two abstract memory locations: the stack and the heap, or it could consider the heap as a single location, and each call frame as another. On the other hand, a more fine-grained approach could take context information into account: instructions that allocate memory could be considered separately in different calling contexts, parametrised by a particular "stack depth" to account for.

7.6.2 Context and Flow Sensitivity

An alias analysis can be global, computing a single conservative "may point to" solution for the entire program, meaning that for a particular pointer, it will never "fall out" of its points-to set for the entire lifetime of the program. In many cases, this analysis will be overly pessimistic. To that end, there are two common ways in which to refine this global view. One computes distinct solutions for each call-site (this is called context sensitivity), and another for each control-flow location (this is called flow sensitivity). These two are somewhat orthogonal: an analysis can be neither, one of them or both. Note that context sensitivity of the analysis itself, and the context sensitivity of abstract memory locations are not the same thing, although usefulness of context-sensitive memory locations is predicated upon a context-sensitive analysis.

7.6.3 Alias Set Representation

As outlined above, the analyses produce a significant quantity of data and could require fairly long time to do so. Moreover, it is desirable that this data be readily available to external tools. As such, providing analysis results in a well-defined format embedded inside LLVM bitcode as metadata which standard LLVM libraries and tools can read seems to be a good compromise.
The structure of the persistent metadata needs to be such that it can easily represent results of multiple different alias analyses, with various degrees of context and flow sensitivity. In particular, this means that the metadata needs to be able to attach points-to sets to particular contexts, whether that context is derived from flow sensitivity or from callsite/callgraph sensitivity.
While top-level variables in LLVM are in an SSA form (meaning that they never change their value during their lifetime), this is not true of allocated memory, whether with `alloca` instructions or with `malloc` (and it wouldn't be necessary to perform further alias analysis if it was). As such, the top-level points-to sets (attached to top-level pointers) never change during their lifetime – however, the points-to sets available indirectly through them do. To take advantage of this fact, it is desirable to decouple transitive points-to relations, so that top-level points-to sets can be attached to definitions of top-level values (i.e. to the instruction that defines them). As such, they are entirely

immutable. The points-to sets can only point at abstract memory locations, top-level variables in LLVM never have their addresses taken and as such no pointers can point at them.
Apart from top-level points-to sets, the metadata needs to also represent sets that are attached to abstract memory locations. These sets will be (depending on the type of the analysis) different at different point in the program, and hence it needs to be possible to keep multiple such sets for each abstract memory location, distinguished by context. The representation of the context needs to be such as to be able to efficiently decide if a particular callstack falls under the given context. For flow-sensitive queries, only the topmost callframe is relevant, while for context-sensitive queries, all but the topmost callframe can be potentially relevant.
The contexts form a semi-lattice, and so do the points-to sets associated to those contexts. Meets of contexts correspond to joins of points-to sets and vice-versa. Intuitively, the broader the context the bigger (more over-approximated) the points-to sets it entails. The idea here is that for a particular context, we can obtain over-approximate points-to set as a union of points-to sets in all contexts that are more specific. Conversely, for a more specific context, if exact information for that context is not available, a points-to set for any enclosing context is a sound over-approximation of the desired answer. This gives us a good way to represent the context → points-to set mappings for abstract memory locations: each abstract memory location has a context tree attached to it.
There are two types of context tree nodes, internal and leaf nodes. An internal node only has a single instruction pointer in it, always pointing to a callsite (there is an additional virtual callsite that represents the entire program, i.e. the root of the static call graph). Leaf nodes additionally contain an actual points-to set. The points-to set for an internal node can be computed as an union of all its children's points-to sets, and as such is not stored explicitly. The depth of a context tree is entirely the discretion of the analysis in question, and queries are satisfied by giving the most-specific points-to set available in the context tree. (In the degenerate case of a context-insensitive analysis, the context tree is singleton.)
To represent flow sensitivity in the data, context-trees are rather cumbersome and inefficient, as they would need to allow representing instruction spans, and for context-insensitive, flow-sensitive analysis, addition of many redundant internal nodes to the context tree. Instead, a different representation is used to represent flow sensitivity, orthogonal to context sensitivity. Recall that instructions have a static points-to set attached to them, representing the *result* of that instruction. Additionally, we can attach a map from abstract memory locations to context trees to each instruction, containing all the AMLs that any of the static points-to set in any of its *arguments* refer to. The context trees in this map then represent the points-to sets for those memory locations at the particular spot in the control flow graph. Again, for context-insensitive, flow-sensitive analyses, these context trees will be singleton. Conversely, for flow-insensitive, context-sensitive analyses, they will just point to the global context trees for the relevant memory locations (over-approximating the flow sensitivity away).

7.6.4 Mapping Alias Sets to LLVM Metadata

To summarise the discussion above, we will recount the various types that come into play in representing points-to sets.

AML (abstract memory location) a unique representation for (a set of) memory locations

PTS (points-to set) a set of abstract memory locations

context tree each node points to a particular callsite, representing the static callgraph of the program; leaf nodes additionally contain a points-to set; representing context trees is particularly tricky, because it is not possible to directly store references to instructions in global metadata; instead, context tree nodes are referenced from callsites, and value use-def chains can be used to look up the callsite for a particular context tree node

AML map a function from abstract memory locations to context trees

instruction each instruction has a single points-to set attached, representing the points-to set of the (top-level) result of this instruction, and a single AML map, which has entries for each element in all static points-to sets for all operands of the instruction, and transitively for all AMLs that arise in points-to sets of any already included AMLs

global PTD (points-to data) a single top-level AML map with entries for all AMLs that exist in the program; can be automatically summarised using results of a flow-sensitive analysis by unioning AML maps attached to all instructions, or can be obtained directly as a result of a flow-insensitive analysis

Recall that LLVM metadata is structured as a graph with labelled edges and nodes being tuples of primitive values (cf. **Section 4.1.6**). The basic idea of our format is to represent the above data types as LLVM metadata nodes. Abstract memory locations are simply represented by nodes carrying a numeric ID. Points-to sets are represented by a single node, with outgoing edges for each element of that set. Context trees are mapped naturally to LLVM metadata trees, AML maps are represented as a list of tuples (AML pointer, context tree pointer). Instructions get two named metadata slots, `!aa_def` and `!aa_use`, first representing the result points-to set and the other an AML map.

7.7 Per-Value Abstraction

We can view program semantics in terms of a number of concrete domains (each corresponding to a particular data type) and an algebra for each such domain, encompassing the operations of the data type available in the programming language. In cases where different concrete domains interact, special care needs to be taken though. There are a few examples of such inter-domain interactions:

- most programming languages contain some sort of boolean concrete domain and a number of predicates over other domains that produce results in the boolean domain
- array dimensions are usually specified using a scalar from some particular concrete domain
- various casting operators: bitcasting, width extension, truncation, etc.

Such interactions cannot be easily captured in isolation for each domain separately. Apart from those limitations though, an abstraction is, in many cases, essentially a homomorphism from the concrete domain's algebra to the abstract domain's. We would like to be able to implement abstractions primarily in terms of such homomorphisms (these correspond to α, the abstraction function, from **Section 2.4.3**). Some attention to inter-domain interaction is unavoidable, but should be kept to a minimum – at least as far as correctness is concerned. It is, however, acceptable for special cases where efficiency or precision improvements are desired.
For the boolean domain, the easiest approach is to fix a *true*/*false*/*maybe* (tri-state logic) "abstract" domain: all abstractions are then expected to map abstract boolean predicates onto either the tri-state abstract domain or the concrete boolean domain of the program (the latter only being possible in fairly special cases).
Casting is more tricky: combinatorial amount of code is required for translating between abstract domains. The simplest approach is to treat bit-casts as a type of aliasing and force both variables to be abstracted into the same abstract domain. This requires that an abstraction for a particular abstract domain can handle all concrete domains in a casting-alias set. We do not expect this to be a problem in practice. Inter-abstract-domain translation code then only needs to be provided in cases where extending the alias sets through casting would lead to unacceptable coarsening of refinement control, or where an abstraction is unable to handle all types (concrete domains) that can appear in a casting-alias set.

7.7.1 Implementing Per-Value Abstraction

The primary use case is producing LLVM bitcode which is as close to the usual LLVM semantics as reasonably possible. In particular, stock LLVM tools should be directly usable with the transformed bitcode. Provided a suitable implementation of non-deterministic choice (random, external test vectors, ...) it should be possible to generate native code of the abstracted program and directly execute it.
With suitable support code, an abstraction can be implemented in terms of a relatively straightforward "lowering function": code that, given a single LLVM instruction with concrete parameters and results, produces equivalent code (not necessarily a single instruction, intermediate values and even non-trivial control flow is permissible) over the abstract domain: the code obtains names of registers that contain the abstract parameters and a register where the abstract result is expected in subsequent code. Such a lowering function is substantially easier to implement than a full transformation

pass. Based on this lowering function, we can then construct a full transformation by supplying code to decide which registers and memory locations hold abstract values in a particular abstract domain and to adjust control flow to reflect the tri-state results of some abstracted operations.
The main limitation of this approach is that the abstract values must have run-time representation that can be implemented using scalar values in some concrete domain available in LLVM. For many abstract domains, this is not a problem but e.g. set-based abstract values cannot easily do this. In cases where a more complex representation for abstract values is required, the desired result can still be achieved using lowering alone, but a small concession in the semantics of the abstract code must be made: the lowered code can use pointers to represent the abstract values, which are then stored in the malloc heap. A compromise needs to be made between allowing to introduce leaks into the bitcode, or having to dutifully allocate and copy abstract values on (nearly) every instruction. We believe that "leaking" the memory of abstract values is acceptable, as it can be done in such a way that downstream tools will be able to easily distinguish such intentional leaks from genuine errors.

7.7.2 Encoding Abstract Values in LLVM

In order to encode abstract values as LLVM scalars, we can leverage the existing type system of LLVM, particularly its ability to create cartesian (or aggregate) types. First, this allows us to easily identify and distinguish abstract values: they will always be of particular specially designated types, say:

```
%lart.interval.i32 = type { i32, i32 }
```

The above LLVM fragment defines a new type, named `%lart.interval.i32`, as a pair of 32-bit integer values. We will use the `%lart` prefix for all types that represent abstract domains. Next, the middle part of the name, `interval` designates which abstract domain this is in particular, and the final component relates the original LLVM type to which this abstract type corresponds. Since aggregate types are distinct from other types of the same bit-width, we can use the builtin LLVM type-checker as an additional layer ensuring the consistency of the abstraction. This prevents mistakes where we would accidentally use an abstract value as concrete or vice-versa: the type system requires explicit conversions to be inserted, and it is easy to ensure that bitcasting to or from abstract types never happens.

7.7.3 Branching and Value Restriction

When a branch is taken due to a *maybe* result of a boolean expression, the effect of the branch can be, in addition to its usual concrete semantics, to restrict the values of some abstracted variables (namely those appearing in the boolean expression that caused

the branch to be taken). Abstract interpreters naturally take advantage of this fact, by building up *path conditions*.
A static abstraction engine can take advantage of the same principle, by restricting values based on which branch has been taken after an indeterminate comparison result. Consider that we have an abstract value x and a conditional branch is taken, based on the comparison $x < 5$. Now if we don't know anything at all about x at the point of comparison, it is clear that if the conditional branch has been (non-deterministically) taken, the value of x must have been less than 5. Likewise, when the branch was *not* taken, the value must be 5 or greater. We can take advantage of this fact by adjusting the abstract value right after entering the relevant basic block (this may require tweaking the relevant φ nodes as well). This basically means intersecting the relevant value with the constraint which enabled the current branch to be taken. Clearly, not all abstract domains can leverage all such data, but the effect can be very useful when they do.
The value restrictions inferred from control flow can be computed independently of particular abstract domains, as long as each abstract domain provides the right set of primitive operations. Namely, it must be possible to compute a new abstract value from an abstract value and a reference to an instruction implementing a predicate in LLVM, such as `icmp`, along with the actual result of that instruction (i.e. *true* or *false*).
In previous section, we have established a naming convention for types that represent abstract values in LLVM bitcode. We will use a similar scheme for naming operations on these types, which are implemented as function calls (similar to LLVM-native "intrinsic" operations like `smul.with.overflow`[52] We will call the particular operation used in value restriction *assume*:

```
%x = call %lart.interval
          @lart.interval.assume(%lart.tristate %cond,
                                %lart.tristate { i2 0 },
                                %lart.interval.i32 %a)
```

meaning that value `%x` encodes a value `%a` restricted to the inverse of the condition encoded in the `icmp` instruction referred to via `%cond` (it is inverse because we are assuming that the actual value of `%cond` is 0, i.e. *false*). We expect that `%cond` is defined like this:

```
%cond = call %lart.tristate
             @lart.interval.icmp.sgt(%lart.interval.i32 %a,
                                     %lart.interval.i32 { 0, 0 })
```

(recall that LLVM bitcode is in a single-static-assignment form, i.e. the result of this `icmp`-like intrinsic call is the final value of `%cond` in this particular context). The *assume* call from earlier then statically asserts that the value of `%cond` is, at this point, 0. A later

[52] See section "Arithmetic with Overflow" in [115], also available online at **http://llvm.org/docs/LangRef.html#arithmetic-with-overflow-intrinsics**.

transformation of the bitcode, specific to a particular abstract domain (the interval domain in this case) will then translate this assumption into code that computes the restricted value (by taking the intersection of the interval %a and the interval $(-2^{31}, 0)$). While at first glance it seems that extending this scheme to jumps farther in the history of the computation may be a good idea, all the values that could benefit from such an increase in the depth of the restriction have already been dynamically constrained by that path condition. Hence, this would only help if it wasn't the case that $(a \wedge b) \wedge c = a \wedge (b \wedge c)$ – a property we would rather like to avoid with our abstract domains.

7.8 Relational Abstractions

In previous sections, we have discussed an entirely non-relational approach to abstraction – each value has been considered in isolation. While this is a simple and robust technique, we would like to be able to do better, especially in terms of precision. In **Section 7.7.3**, we have established a way to leverage control-flow decisions in improving precision of per-value abstractions. A similar approach can be used to restore a limited amount of relationality into the abstraction, taking advantage of relationships between different abstract values. Specifically, code like $x \leftarrow y$ establishes that $x = y$, from that point on, until something happens to one or the other. Now if it so happens that either of those values is constrained through a control-flow decision, we might be able to derive knowledge about the other as well – and not only in the case that $x = y$ but also in more complex cases, like $x = y + k$ and so on. Hence, even though the fact that $x = y$ is not encoded in the program state as such, in the window where we statically know that this is the case, we can propagate constraints applied to one also to the other (possibly after a suitable adjustment, in more complex cases).
While this does not replace fully relational abstractions – i.e. where a predicate of the form $x = y$ is part of the program state – it constitutes a viable compromise in between. The constraint propagation scheme does not add too much complexity to the implementation and it retains part of the precision benefits of a fully relational abstraction. Since it is computed statically, it also avoids the overhead of introducing relational predicates into the encoding of program states. Finally, even though static constraint propagation approximates relational abstractions, it does not in any way preclude their use: the same program could use constraint propagation for some values (which use strictly non-relational abstract domains) but also use relational domains for a different set of values.

7.8.1 Implementing Constraint Propagation

Besides value abstraction, which can be implemented as an LLVM-to-LLVM transformation pass (or as a sequence of such transformations), another such transformation can be implemented for constraint propagation. The job of constraint propagation, as detailed above, is to insert additional calls to relevant *assume* intrinsics, based not only

directly on the path condition, but also for values with more complicated data-flow relationship to the values restricted by the "vanilla" path condition.
Since, like with all other techniques discussed in this chapter, the changes to be done to the program need to be computed statically, there are some limitations: most importantly, it is not clear how far back to look for instructions to use for generating and propagating constraints. Conceivably, a heuristic would compute the limit based on the actual program text, although we can just as well make the depth a parameter of the transformation. The other limitation is that, clearly, the transformation can only reach within the current static scope: the parameters passed to the function and any global state of the program are entirely unknown and as such cannot be leveraged to create additional constraints. The constraint propagation pass may be, however, combined with an inliner to expand its reach – when a function is inlined, replacing the callsite with the body of the function, its enclosing scope becomes statically available in that instance, and the constraint analysis (and in turn, the final abstraction) can become more precise.
In implementing the propagation pass, there is a trade-off to be made between shifting large part of the work on to the implementation of a particular abstract domain and making the generic part of the transform rather simple, or making the generic code more elaborate (and complicated) and retaining simplicity in the abstract domain. The trade-off centres around how to generate the *assume* calls and their signature. If we re-use the same signature that we have used in "plain" value restriction:

```
%x = call %lart.interval
          @lart.interval.assume(%lart.tristate %cond,
                                %lart.tristate { i2 0 },
                                %lart.interval.i32 %b)
```

then the lowering function for this *assume* call will need to work out the data-flow implications of `%cond` on its own, based entirely on the origin of `%b`. An alternative would be for the generic pass to encode the relationship of `%b` to the immediate parameters of `%cond` into the call itself. However, since LLVM makes use-def information readily available, following the use-def chains of parameters of the instruction which computed `%cond` is not very hard. Eventually, the use-def tree will reach the definition of `%b`. Consequently, the correct spot to draw the line appears to be ensuring that the value of `%cond` only depends on the value of `%b` "straightforwardly" – in other words, that it is not a result of a dynamic construct (loops and function calls, including recursion, would make the requisite data-flow analysis too complicated). As long as the lowering function for *assume* can rely on the data flow between `%b` and `%cond` being simple, this does not constitute a substantial problem for its implementation.

7.8.2 Abstract Memory

An important challenge in abstraction is the abstract memory model to use. The problem statement is similar as in pointer analysis: we need to compute a static approxima-

tion of dynamic memory that is safe (i.e. any behaviour admitted in the actual execution with real dynamic memory is admissible in the abstracted program with approximate static memory). The simplest and most straightforward (non-trivial) model used in pointer analysis can be adapted for use in program abstraction. This model assigns a unique static location to every instruction in the program that can allocate dynamic memory. If the dynamic behaviour of the program creates multiple instances of a particular static memory location, the values stored at those locations must be *joins* of the individual abstract values. This also means that when memory allocation is abstracted using this technique, the value stored at the dynamic location needs to be in an abstract domain with a *join* operation available.

This is clearly an over-approximation, and a fairly coarse one. It is, however, easy to implement and it allows us to abstract away loops that allocate memory. A straightforward extension is to create a fixed-size "buffer", with new allocations getting *join*-ed into a slot corresponding to the index of the allocation modulo the size of the buffer. This is clearly a refinement, but it still admits finite-state (with a known bound) approximation of loops. Again, the values must reside in a suitable abstract domain.

8 Conclusions

In this thesis, we have explored a section of the large and diverse landscape of formal verification. Our main focus was on exploiting the form in which most programs exist: source code, which is automatically translated into machine code. For many programs, their source code is their only description that remotely approaches completeness, and in 99 % of cases, it is their only *formal* description. We expect that this will remain to be the case well into the future. This fact is what justifies the effort funnelled into research around verification of software in its, may we say *natural*, form. The goal of the programmer is to strike a balance in their program between talking to the machine and talking to other humans. Many new programming languages strive to improve this balance on both sides simultaneously. As time goes on, source code is becoming both a better communication medium and a better substrate for automated verification. We can but hope that our work has contributed usefully to this wider stream within computer science research.

Accordingly, the main body of this thesis comes in the form of source code. As it is appropriate for technical papers to be available for consultation to the broadest possible audience, the same holds for program source code. It is, after all, (in some sense) the most faithful way to express ideas. Hence, we have chosen to licence the source code in a way that allows it to be read, quoted and re-used in the efforts of others.[53] Of course, the code can be easily obtained on the Internet, from **`http://divine.fi.muni.cz`**.

8.1 Contribution

We believe that this thesis has contributed important and useful knowledge to the state of the art in software verification (and, just as importantly, error discovery). The contributions can be broken down into a few areas, roughly corresponding to the chapters 2-7 of the thesis. Additionally, we believe our summary of the state of the art in **Chapter 2** is valuable in itself, drawing upon multiple areas of computer science that are not always considered together in the context of model checking.

Code First and foremost, we would like to re-iterate that our most important contribution comes in the form of code, which cannot reasonably be printed or otherwise published in the traditional sense. The source code of DIVINE represents a formal, mechanised description of all the topics discussed in this thesis. Moreover, while DIVINE is a high-performance tool and it has to make concessions to the hardware platform, we believe that it is concise and comprehensible to humans just as well. You could treat the text of this thesis as the "Cliff's Notes" version of DIVINE.

Parallelism We have introduced, verified correctness and quantified the performance of a number of crucial building blocks: algorithms and data structures. These building blocks allow

[53] Most of the code comes under a 2-clause Berkeley Software Distribution licence. The full text of the licence is available alongside the source code itself.

us to retain a high level of abstraction in the description of model checking algorithms in DIVINE, while at the same time offering superior performance. The building blocks also provide great potential for re-use in other endeavours, outside of model checking.

LLVM We have shown how to build an explicit-state model checker for LLVM. We have considered how to represent memory and how to treat pointers, how to deal with exception mechanisms often used in higher-level programming languages. We have also extensively dealt with the support code needed by real-world programs: language runtimes and system and standard libraries; and with the interface between the model checker and the program being model-checked (analogous to the interface that the operating system provides to system libraries).

Properties We have classified the properties commonly sought in programs and looked at how these map to model checking as provided by DIVINE and especially its LLVM support. We explored the combination of LTL properties with software (as opposed to models), a topic largely ignored in the literature. We introduce a novel approach to specifying such properties succinctly and intuitively.

Reductions We have introduced a new reduction ($\tau+$) tailored to explicit-state model checking of LLVM bitcode with parallelism. We have also described an implementation of heap symmetry reduction in an environment with unrestricted, untagged pointers. Moreover, we have presented an improvement of the pre-existing *partial order reduction*, suitable for parallel search algorithms.

Abstraction Finally, we have proposed a novel approach to abstraction in the context of LLVM, as a composable program transformation. This provides us with a framework to easily implement and evaluate various abstractions in a real-world context. Our approach, just as importantly, closes the gap between the way we think about abstractions and the way they are implemented in practice in model checkers.

8.2 Future Work

Most software projects that are in active use are also in active development. In this regard, DIVINE is no exception: it could hardly be considered complete. There are many ideas on how to extend and improve its functionality – some are a "simple matter of coding" while others need some theoretical groundwork. In either case, the list provided here is far from exhaustive, it merely lists the ideas that we would like to see implemented over the next few years.

Search While search algorithms have been the "founding topic" of DIVINE many years ago, the space of possibilities is far from exhausted. Heuristic and bounded searches could improve bug discovery rate for large inputs, and bounded searches could provide weaker guarantees about some properties even in cases where a full search cannot be done. A prime candidate is the number of context switches that happened in the system, as the majority of counterexamples have fairly few. Another application would be to speed

up abstraction/refinement loops by using the spurious counterexample from previous iteration as a guide.
For systems where computing the transition function is fast (this is not the case for most LLVM inputs, but in many other applications of DIVINE it is), trading reductions in memory use for extra computations may be a worthwhile goal.
While the compression scheme currently implemented in DIVINE is certainly quite effective (and efficient), it barely scratches the surface of the possibilities to be explored. Likewise, "lossy" compression (hash compaction and the like) has a lot more potential than the current implementation in DIVINE taps.

LLVM Without doubt, the LLVM interpreter in DIVINE is the most complete in any currently available model checker. Nevertheless, there is still room for improvement. One sorely missing piece is a register allocator: the current code allocates a distinct slot for each LLVM register, which is, in some cases, extremely wasteful. Even if tree compression can somewhat offset the problem, it is still a major issue. Moreover, state canonisation takes time linear in the size of a state, which means reducing state size will dramatically improve performance.
There are a few minor features missing, although now that exception handling is fully functional, these all fall into the "simple matter of code" category. Nevertheless, the code eventually needs to be written.
Finally, one major feature is missing in our implementation: memory models other than sequential consistency. Unfortunately, even implementations based on store buffers with a small bound on their size cause a substantial state space blowup. Therefore, a practical implementation will likely need some research into how to offset this blowup.

Abstraction In the context of DIVINE and its ecosystem, abstraction is the area with most work left. The ideas presented in this thesis undoubtedly constitute a very promising direction. Nonetheless, we do not know for certain, and there is only one way to find out.

A C++ Bricks

In this technical appendix, we will describe some of the reusable C++ code that has accumulated over the course of development of DIVINE. While a high-level overview of *some* of the code was given in **Chapter 3**, many aspects were not covered there.

The `bricks` C++ library was designed to make code re-use as easy as possible. Since C++ lacks a formal package system, this puts considerable constraints on how to technically achieve code sharing. On one hand, external packaged libraries are the usual way to redistribute code for re-use in multiple programs. This comes with considerable downsides though: for a C++ project, each external dependency, especially a hard dependency, constitutes an extra hurdle for the end-user who would wish to compile the program from source. Just as importantly, it is inconvenient for developers – they need to install the library and keep it up to date. Things become even more problematic when a change in the library is required for the application to work correctly. If the library is in a separate package from the program, this causes a lengthy release process for the library and a dependency change for the program. All downstream users and developers then need to update the library to a correct version. When adding code to the library, even when the changes are temporarily confined to the development machine of a single person, the edit-build-run cycle becomes needlessly more complicated when multiple packages become involved.

All in all, the approach with a fully external library is only suitable in cases where the code-base is very mature and stable and where there is no tight integration of the library code with the program. Another approach is *embedding* the shared source code in multiple projects. This clearly has downsides as well: multiple projects will contain multiple copies of the shared code. In the conventional world of libraries that more often get transparent bugfixes, this is the worst possible solution: each program that uses the library needs special attention every time a problem arises in the library. This is especially important in security-sensitive applications.

Nevertheless, there is a narrow area of applications where embedding is, despite its shortcomings, the superior alternative. The possibly most important aspect of a setup where such a configuration can work is rigorous version control: in practice, the different copies of the shared code become version control branches. This allows changes to be moved from one such branch to another with ease, even if the branches have divergent sets of changes. The other aspect is the nature of the library code in question: in our case, it is mainly utility code – data structures, algorithms and so on.[54]

With those limitations in mind, we have chosen to package up the `bricks` as C++ headers, single header per a logical unit. Each unit comes with its own unit tests which are part of the unit header and some also come with performance benchmarks; the

[54] As hinted above, *cryptographic* algorithms and primitives constitute a special category. Any security-sensitive code shared among multiple packages should be well separated from all of them and maintained as a distinct package. None of this applies to code in the `bricks` repository.

results of some of these are presented in **Appendix B**. Most of those units (or bricks) are C++11 code and cannot be compiled with older (C++98) compilers.

A.1 Heterogeneous Lists

While in dynamically typed languages, the list type is usually heterogeneous by default, in statically typed languages, collections normally have uniform types, and lists are not an exception. Nevertheless, it is often very useful to be able to build a collection of entities of distinct types. Other times, it is useful to be able to build a collection of *types* themselves, without any data – this is especially true for type-level metaprogramming. In DIVINE, for example, the template instantiations for a particular combinations of an input language, an algorithm and other components involved in graph exploration are built using a metaprogramming framework. The bricks repository contains two distinct heterogeneous list implementations, one is for purely type-level programming and represents no runtime values; this is available as `brick::hlist::TypeList`. Usual list operations are available at compile-time, including list concatenation, map, filter and so on.

The other heterogeneous list implemented is a runtime data structure, where both the size and the type of each element is known statically (at compile time). The elements are compactly represented in memory as a contiguous block. This data structure is available as `brick::hlist::Cons` (after the LISP operator cons for making a list cell). The main use-case for `Cons` is type-level recursion, as used in, for example, the implementation of the LLVM interpreter's instruction dispatch routine (in `divine/llvm/execution.h`).

A.2 Remote Procedure Calls

When implementing distributed software, there are many options how to implement communication between the various processes. At the lowest level, there is always some sort of a message-passing interface. The standard way to implement message passing in a cluster is through the MPI specification [118]. However, MPI is data-oriented, and does not allow for control constructs to be distributed across multiple nodes. In DIVINE, parallel control is achieved through a few simple primitives: `distribute`, `collect` and `parallel` – first distributes a value by issuing a method call with a parameter in each thread, the second collects return values from a method invoked in a similar way and finally `parallel` just runs a method in each thread. Additionally, in the distributed-memory mode, a `ring` method of execution is required: a method is executed in each thread sequentially, passing an accumulator from one to the next. This is required when the method in question needs to access data that is only available locally to each thread (which can be the case when different threads run on different machines in a cluster, eg.).

Implementing these primitives is simple in a threaded model, with shared memory: the parameters, threads, etc. share program text *and* all data – pointers, including function pointers and member-function pointers, are equally valid across all threads. Hence, all

of the primitives can be implemented in terms of passing a member-function pointer to each thread and letting it execute it. In a distributed system, however, function pointers from one node are not necessarily valid on another node – heterogeneous clusters, address space layout randomisation, slightly different library or program versions or builds running on different nodes could all cause function pointers to become invalid when copied from one node to another.
For this reason, passing control over MPI messages needs to be implemented using a dispatch mechanism. Function pointers need to be translated to portable integral identifiers and function call parameters need to be packed into platform-independent bit vectors. While implementing such a dispatch mechanism manually is straightforward, it is extremely tedious and somewhat error-prone. Message IDs need to be carefully allocated and the native mechanisms for control flow (most importantly function calls) cannot be directly used for invoking code on remote nodes.
The `brick::rpc` unit contains code to automate the translation of method calls to portable identifiers and back again to function pointers on the remote end. The only requirement is that each class participating in remote procedure calls enumerates the methods that need to be available for remote calls using a special `BRICK_RPC` macro in its class definition. This is necessary so that method identifiers can be allocated. The available methods for each class are accumulated at compile time in a heterogeneous list (see **Section A.1**), also across class inheritance hierarchies. An identifier ↔ function pointer mapping is established at compile time and the `rpc::marshall` and `rpc::demarshall` methods can be used to translate a method invocation, along with parameters, to a bit vector that can be sent across to another machine using MPI or any other message-passing mechanism. The `rpc` calls are further abstracted in DIVINE to make the remote calls entirely transparent in algorithm implementation.

A.3 Tagged Unions and Algebraic Data

In functional programming languages, algebraic data types (ADTs) are the norm. In C++, product types are readily available in the form of the `struct` construct, or even in form of classes (also known as record types). Unfortunately, sum types (tagged, or discriminated unions) are not available as part of the language and consequently, neither is pattern matching on those. In C++11, though, it is possible to implement a tagged union with a limited form of pattern matching using the C++11 lambdas. The bricks repository contains such an implementation, along with simple pattern matching. The bricks implementation could be more appropriately called a type-tagged union, since the tags of the different data constructors correspond 1:1 with the types of the fields. A simple algebraic data type implemented using our type-tagged unions might look like this:

```
struct A { int x; };
struct B { int x, y; };
type::Union< A, B > AorB;
```

The definition above is roughly equivalent to the Haskell definition below. The disadvantage of the C++ version is that it is more verbose, but it has advantages too, in that the product types can be more readily accessed outside of pattern match context.

```
data AorB = A Int | B Int Int
```

We can then assign either A or B values to variables of type `AorB`:

```
AorB x;
x = A{ 0 };    ASSERT( x.is< A >() );
x = B{ 1, 2 }; ASSERT( x.is< B >() );
```

and more interestingly, we can write simple pattern matches on them:

```
x.match( [](  A a ) { std::cout << "A: " << a.x; },
         [](  B b ) { std::cout << "B: " << b.x << ", " << b.y; } );
```

There are many use-cases for algebraic data, the most common is probably for a concise representation of abstract syntax trees. In DIVINE, ADTs are used in the implementation of the SILK language parser, see also **Section 5.3.1**.

A.4 Recursive-Descent Parsing

Parsing is a very common task in computer programming: more complex program inputs are usually structured and conform to particular format rules. Such rules are often described using context-free grammars, and the procedure for matching text against the rules of a grammar and constructing a derivation tree (or more usefully, an abstract syntax tree) corresponding to the input is called a parser. Clearly, it is possible to write a parser for a particular language by hand, as a program. However, writing a grammar is usually easier than writing an unstructured program to construct a syntax tree. Therefore, it is customary to implement parsers that are structured around context-free grammars of their corresponding input languages.
Tools such as `yacc` and `bison` help with construction of such parsers, using the concept of attribute grammars – a grammar annotated with program fragments to construct appropriate outputs. In early days of computing, it was often desirable to produce output "on the fly", in step with processing the input, to conserve memory. Attribute grammars have been an useful tool in this regard, making it possible to automatically derive an entire compiler from the attribute grammar, with some clever use of backpatching to "fix up" code generated earlier based on data coming in late in the process (the address of the "else" branch, forward goto,, etc.). Nevertheless, apart from very specialised applications, computer memory has been comparatively cheap and memory is many time larger than a typical source text. Contemporary parsers therefore construct an abstract representation of the source code in memory, using an abstract syntax tree,

and the later phases of processing manipulate this abstract syntax tree. This approach is much more convenient from the point of view of the programmer.
Clearly, attribute grammars can be still used to write parsers which in turn construct abstract syntax trees. There is however another limitation: the grammar needs to be context-free, and to be able to automatically derive an efficient parser, it often needs to be in a particular restrictive form, such as LL(k) or LR(k) for some small value of k. An alternative is to construct parsers using the *recursive descent* approach, where the grammar is not written out as such, but is encoded in the structure of the parser. Non-terminals of the grammar correspond to procedures which read from input and produce a node in the abstract syntax tree as their result. The rules for a particular non-terminal are encoded in the procedure body, reading terminals from the input stream and/or calling other procedures that correspond to other non-terminals. The parser that corresponds to this implementation style is in an LL form in some sense, but is not restricted to context-free parsing. Since the parser is directly written in a Turing-equivalent language, deviations from context-freedom are easily encoded, especially since the previous context is often available in the form of a (partially constructed) abstract syntax tree. Arbitrary look-ahead is clearly possible, even if it is often inefficient.
The bricks repository contains helper classes to make implementation of recursive-descent parsing as simple and convenient as possible. To a certain degree, the primitives provided by the parsing library are inspired by the *combinator parsing* approach, which is the functional variant of recursive-descent parsing. The `parse::Lexer` class provides a simple lexer[55] which can be optionally used to process the input stream. The `brick::Parser` class then serves as a base class for non-terminals: while in procedural languages like C, recursive descent parsers usually use procedures for non-terminals, in C++ it is more convenient to use a class in their place. This arrangement also makes it possible for the classes that represent non-terminals to also double as AST (abstract syntax tree) nodes. Compared to an approach with attribute grammars, this gives a much more streamlined design, since the grammar only exists in one copy – the parser and the AST are derived from the same source. In a conventional `yacc`-style parser, a design with AST would require the AST to be defined separately from the attribute grammar, essentially requiring the programmer to manually keep these two parallel grammar definitions synchronised.

A.5 Bit-Level Operations

Since DIVINE is often extremely memory-constrained, it makes perfect sense to optimise data layout down to smallest details. This often requires bit-level memory access for tightest possible packing of information. Unfortunately, even though C++ provides bit fields, the exact layout of those is unspecified and left for the compiler to decide,

[55] Customarily, the terminals of the language which a parser processes are not letters and symbols of the English alphabet. Instead, a simpler, regular language pre-processor is used to extract meaningful symbol combinations from the input stream, corresponding to keywords, operators, constants and so on. This pre-processor is known either as a tokenizer or a lexer.

possibly on a platform-by-platform basis. This means that bit fields are not suitable in the cases where exact control over the layout of the bits in memory is required. To this end, the bricks repository provides class templates for fully custom and fully specified bit-level tuples. This allows for construction of data types like the following:

```
bitlevel::BitTuple<
  BitField< std::pair< uint64_t, uint64_t >, 120 >,
  BitField< uint64_t, 40 > > x;
```

In this example, the bit vector contains a pair of 64-bit numbers shortened to 120 bits and a single 64 bit numbers shortened to 40 bits, for a total of 160 bits or 5 32-bit words. The product types constructed this way admit an indexing operation (using numeric offsets) which provides a proxy value that behaves like a reference for easy manipulation. The implementation of the `BitTuple` type ensures that all access to a given field uses correct shifting and masking to map to the correct bits in memory.
In addition to tightly packing data, the exact layout of `BitTuples` allows for a straightforward implementation of single-bit spinlocks on data exceeding the length of a machine word. This means that data structures with a single spare bit (this is often the case whenever they contain an integer value which does not need full 32 or 64 bits, or a pointer with certain known minimal alignment) can be locked cheaply by leveraging the `BitLock` class (which can be used as a field in a `BitTuple` and occupies a single bit). In DIVINE, per-state algorithm data is protected by such a single-bit lock when a single shared hash table is in use (and hence multiple threads can attempt to access the same state at the same time).

A.6 Command Line Parsing

Programs with a command-line interface often need to provide complex command and switch combinations. The command invocation needs to be validated and parsed by the program to decide which actions to take. The command interface of DIVINE is relatively complex (see also the documentation in the DIVINE manual [32]), and as such can benefit from code that automates the common tasks of command-line processing. The code in `brick-commandline.h` provides a simple declarative interface for describing command line structure: which sub-commands are available, which command-line switches are allowed for which sub-commands, whether a particular switch takes a parameter and if so, of what data type. It also automates the generation of online help (the `divine help` sub-command) and parameter validation.

A.7 Unit Testing

Unit testing is an important tool for ensuring correctness of interfaces at the appropriate abstraction level. Clearly, unit testing is not very rigorous, but is nevertheless very important in practice. While testing the entire program (functional testing) makes it

possible to uncover unwanted unit interactions, it is usually ill-suited for discovering problems in individual units. The main advantage of unit testing is that the tests are tightly integrated with the code at a relatively low abstraction level – units should come with an interface specification in some form. The role of the unit tests is to check that this specification is actually met by the implementation. In fact, writing an (informal) specification for a unit is often much easier than doing so for the entire program. Moreover, the unit normally only interacts with other parts of the program and only in limited ways (at least in a good system design which obeys the loose coupling principle).
There are two main areas where support code is useful in unit testing: first is organising and registering test cases and providing the framework for running the tests, the other is providing a comfortable way to express assertions that give informative error messages. The approach bricks takes for registering unit tests is to provide a special macro TEST, which is used to define methods that become unit tests:

```
struct TestMyUnit {
    TEST(assertions) {
        ASSERT( 1 == 1 );
        ASSERT_LEQ( 1, 2 );
    }
```

the TEST macro expands to code that takes care of registering the unit test in a central test database. When a test runner is built using classes provided by the bricks unit-testing framework and is compiled with `-DBRICK_UNITTEST_REG`, the test runner will automatically turn all classes which contain tests into suites and make them available for execution. The test runner also ensures proper test separation: each test is started in a fresh process to avoid lingering effects from previous tests (especially undetected heap corruption) and to properly deal with fatal flaws in individual test cases (like a segmentation fault). Typical output of running the unit test suite looks like the following:

```
[  0%] brick_test::unittest::SelfTest ... 3 ok
[  5%]             types::Mixins . 1 ok
[ 10%]                    UnionTest .... 4 ok
[... shortened...]
[ 95%]             rpc::Bitstream ... 3 ok
# summary: 87 ok
```

When we introduce a test failure, the output would look like this:

```
[... shortened...]
[ 95%]             rpc::Bitstream ..
# case brick_test::rpc::Bitstream::_bitstream_64 failed:
  bricks/brick-rpc.h: 380: assertion `x == 2ull' failed;
  got [3] != [2] instead
brick_test::rpc::Bitstream  2 ok, 1 failed
# summary: 86 ok, 1 failed
```

A.8 Functional Testing

While unit testing provides assurances about small components of the system and correctness of their interfaces, functional testing is just as important to ensure proper working of the system as a whole. Moreover, the whole often has functionality that does not come from one particular unit, but is a result of non-trivial cooperation of many different units. In DIVINE, a typical model checking run employs a model interpreter, queues, hash tables, the model checking algorithm itself, possibly a compression or a hash compaction algorithm and so on. The proper interoperation of those components needs to be checked, ideally every time the code changes. The functional test suite of DIVINE is written in `bash` (Bourne Again SHell) and consists of executing the `divine` binary with varying combinations of inputs and parameters, using selected models in various formats with small state spaces.
The tests themselves use `bash` features to detect command failures and a few simple functions to analyse the machine-readable report DIVINE produces after each run. The bricks repository then provides code to run those functional tests in a controlled manner and under proper supervision. Since the functional tests for DIVINE are relatively simple and undemanding, most of the features available in the functional test supervision code available in `brick-shelltest.h` are not used in this particular case. It is, however, used in other projects which require full machine access, gathering of kernel logs, log progress journalling and other such advanced features.

A.9 Benchmarks

Unit benchmarks is another area where registration and running of individual benchmarks can benefit from shared support code. However, while for unit tests the supervision code is quite simple and the most important role of the shared code is to handle test case registration, the supervision of benchmark execution is much more complex. The code in the bricks repository contains code for benchmark registration analogous to unit test registration. More importantly though, it provides benchmark supervision code closely aligned with the requirements and design that were discussed in **Section 3.1.2**, ensuring robust benchmark execution and evaluation.

The code, in addition to reporting benchmark results online as they are executed, provides automated result plotting through gnuplot. In addition to standard gnuplot features, the plotting code in `brick-gnuplot.h` produces plots using optimised colour palettes based on CIElab colour relationships and consistent key/axis colouring across multiple benchmarks (i.e. different quantities use different colour palettes, while the same quantity measured in multiple different benchmarks will always use the same palette). Moreover, it can also provide cubic interpolation to better fit incomplete data (since it is usually not feasible to measure all possible parameter combinations separately). Finally, it produces standalone gnuplot input files, which contain all the data points inline and are therefore easily portable and convenient to work with.

A.10 Others

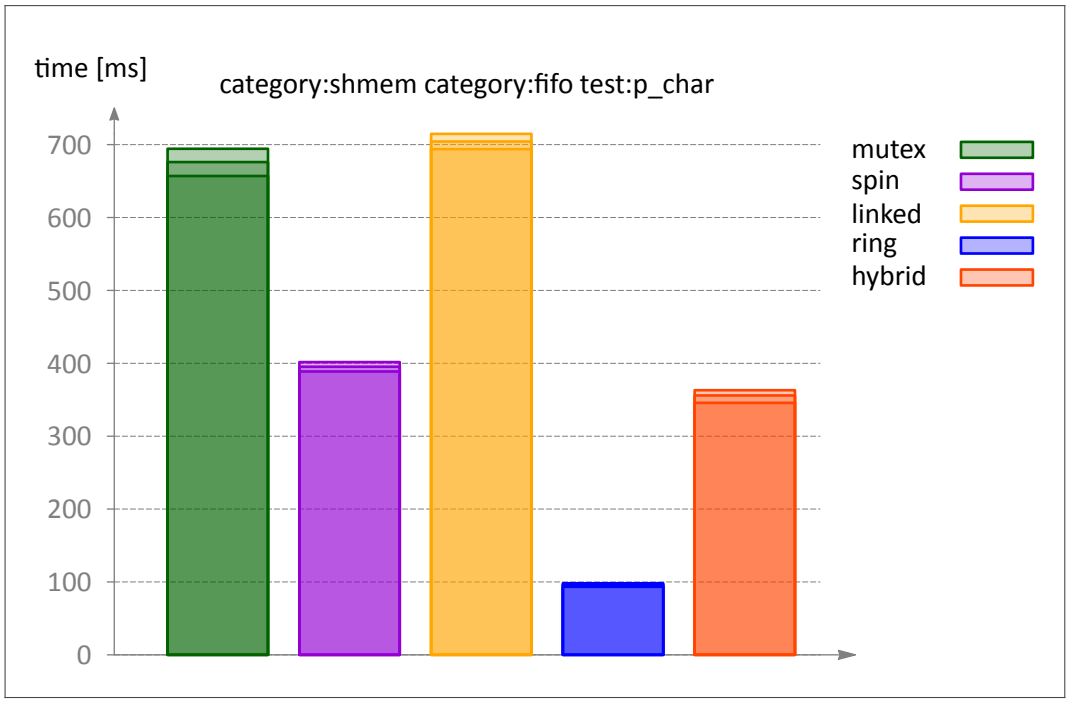
The bricks repository also contains a hash table implementation which we discussed in detail in **Section 3.5**, a shared queue implementation discussed in **Section 3.4**, IPC queue implementation that was detailed in **Section 3.2**. Since all of these have been discussed in detail in the relevant sections, we do not repeat a discussion of those units here.

B Measurement Data

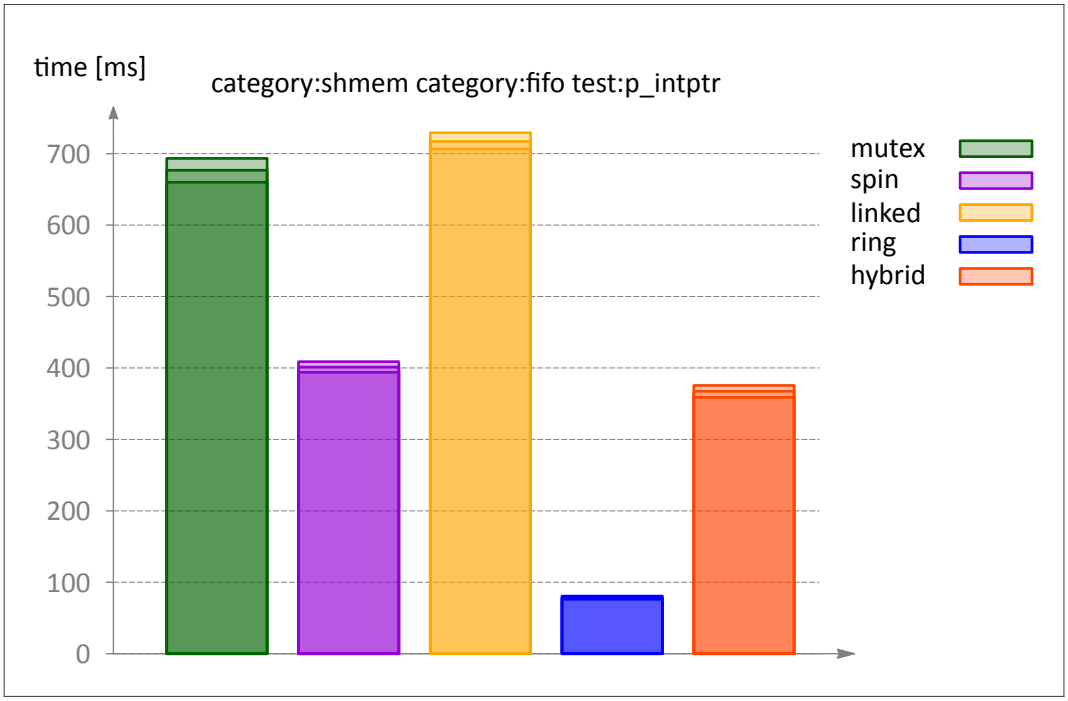
This appendix contains some of the measurement data obtained in various benchmarks. The methodology of obtaining the data was described in **Section 3.1.2**. All plots in this appendix have been produced by the benchmarking subsystem of C++ `bricks`, cf. **Section A.9**.

B.1 IPC Queues

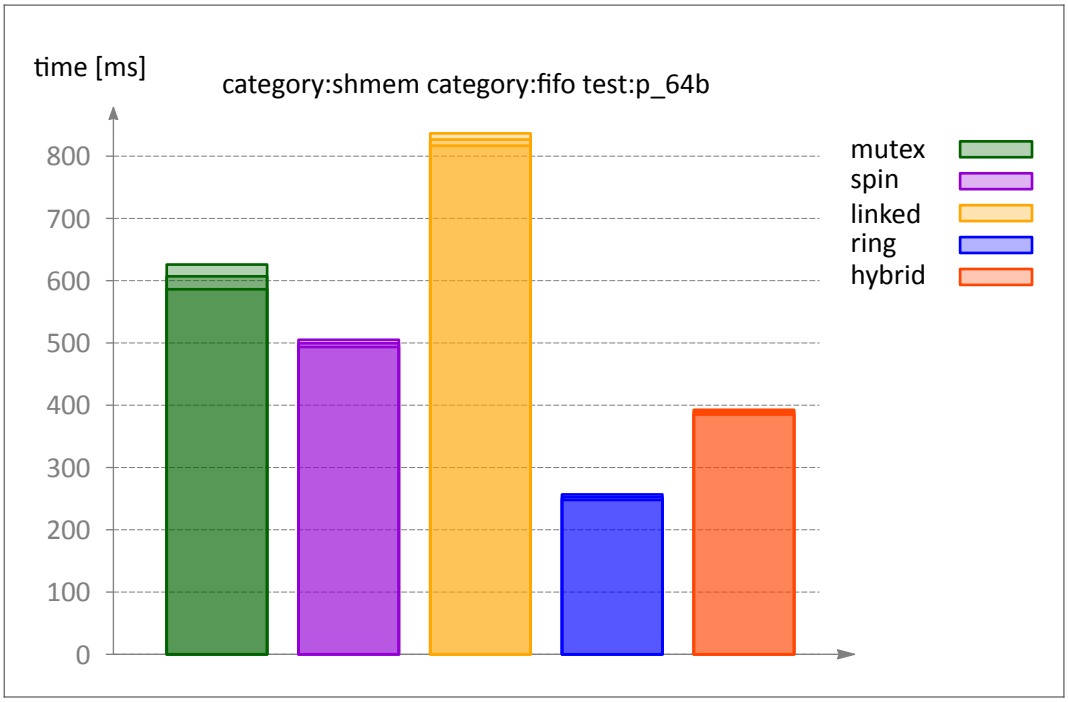
For IPC queues, there is only a limited number of options when it comes to benchmarking. The performance of all implementations is basically invariant under both queue size and the parameters of the queue (within some reasonable limits). Hence, we only include a comparison of the different implementations in a producer/consumer scenario, shifting a million items of a given type, for 3 different types, namely a single byte (**Figure B.1**), a single pointer (**Figure B.2**) and a 64-byte block of data (**Figure B.3**). The *mutex* and *spin* variants are simply a `std::deque` protected by a mutex and a spinlock respectively, the *linked* variant is a lock-free linked list (it is, interestingly, slower than a spinlock-protected deque, presumably for two reasons: its memory locality is very poor, and the emptiness check causes contention on shared variables), *ring* is a ring-buffer (fixed-length) implementation and finally *hybrid* is the design used in DIVINE.



B.1


time [ms]
category:shmem category:fifo test:p_intptr
700
600
500
400
300
200
100
0
mutex
spin
linked
ring
hybrid

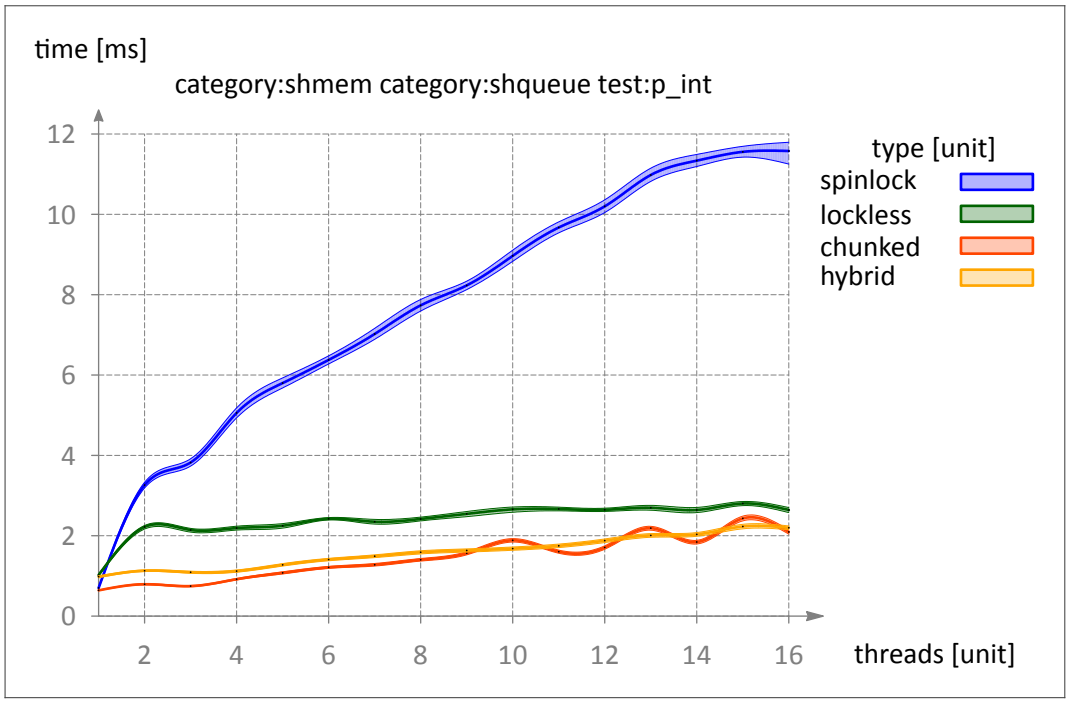
B.2


time [ms]
category:shmem category:fifo test:p_64b
800
700
600
500
400
300
200
100
0
mutex
spin
linked
ring
hybrid

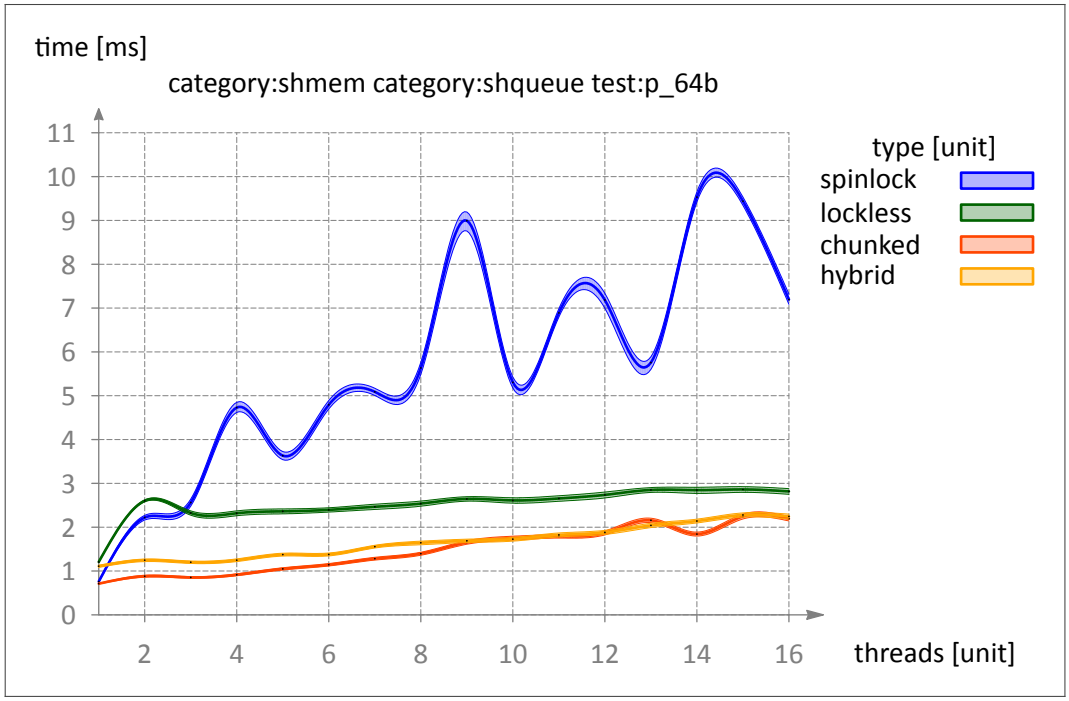
B.3

B.2 Shared Queues

In comparison to IPC queues, the use cases for shared queues are somewhat more varied: most importantly, the number of threads is not fixed, since individual threads access both ends of the queue and a single queue is used by many threads. Our main benchmark scenario then is a single queue shared by n worker threads, each thread adding and removing items from the queue, in a fashion similar to how graph exploration works. Again, we used a few different item sizes, and looked at the scalability (with number of threads) of each implementation. The plot for pointer-sized items can be found in **Figure 3.3**.



B.4



B.5

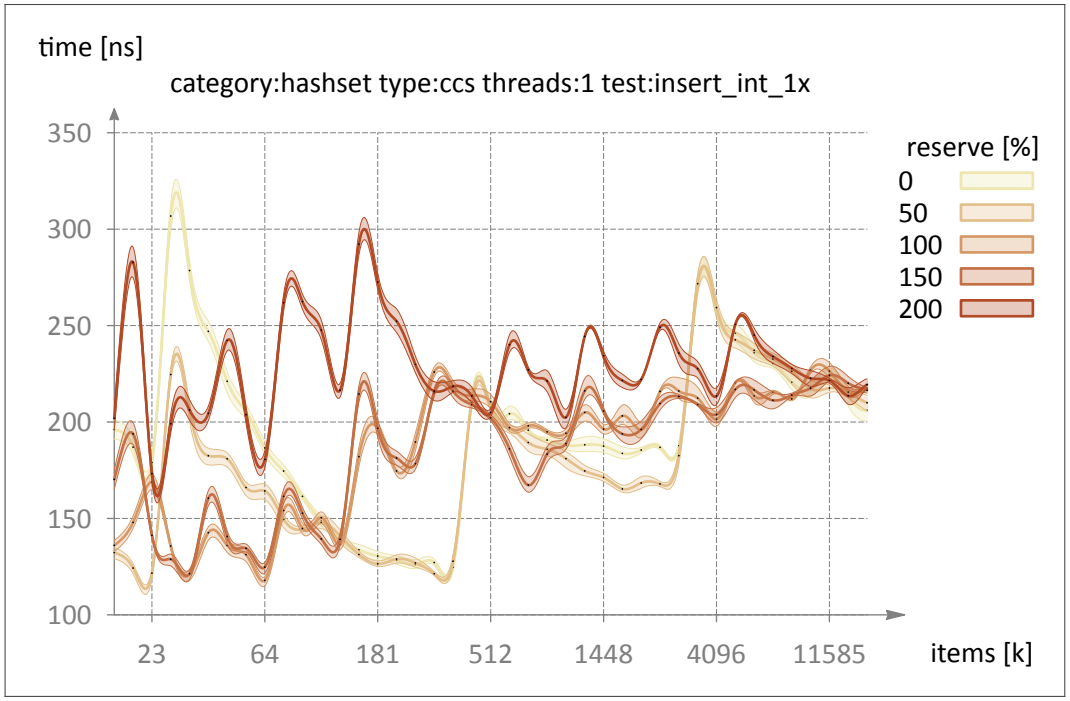
B.3 Hash Tables

This data supplements **Section 3.5.8**. The data is split into a few categories, depending on what is being measured.

B.3.1 Parallel Access and Reserve, Integer Keys

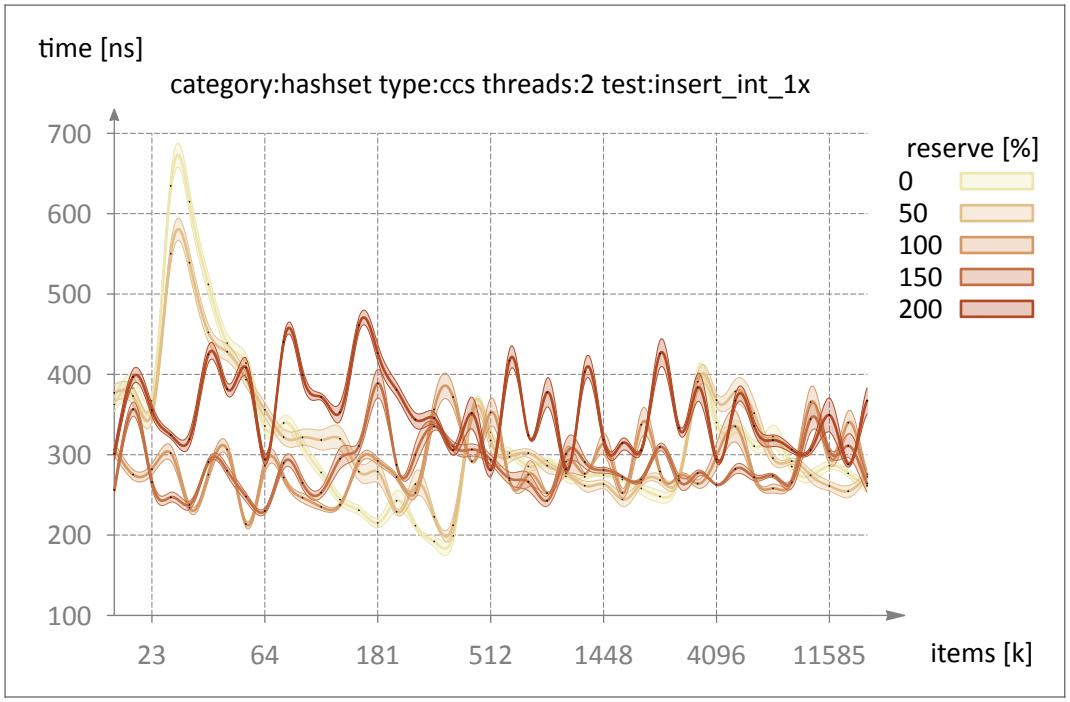
We have measured the effect of creating hash tables with a size that matches the size of the expected data set, as opposed to creating them with a small default size and letting them grow as needed. We have measured the time for inserting elements (no repeats), scaling with hash table size, and with different number of threads. The following two plots are for one and two threads respectively, illustrating that the effect of reserving space flattens out as the hash table becomes bigger, and that for multi-threaded access, the effect of a pre-sized hash table is less pronounced. The “waves” in the plots, on the other hand, illustrate the cost of resizing the table: the highest points of the plot coincide with configurations where a resize is required just before inserting the last few items (and thus the final resize fails to amortise over subsequent insertions).

CCS
1 thread



B.6

CCS
2 threads

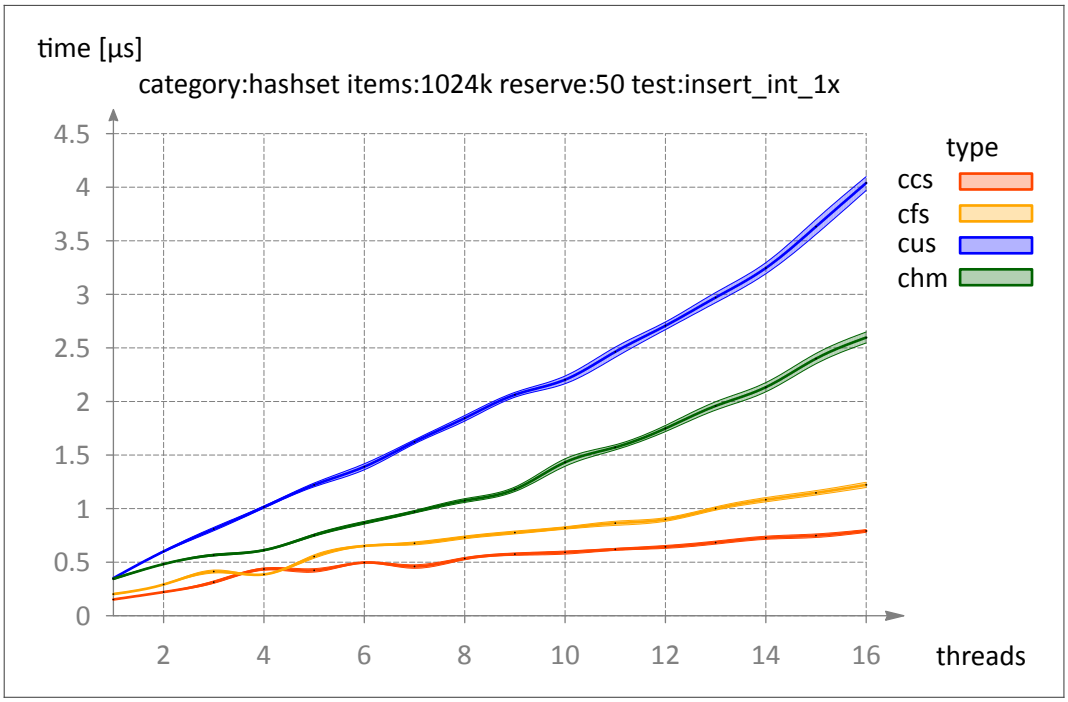


B.7

B.3.2 Scalability of Parallel Access, Integer Keys

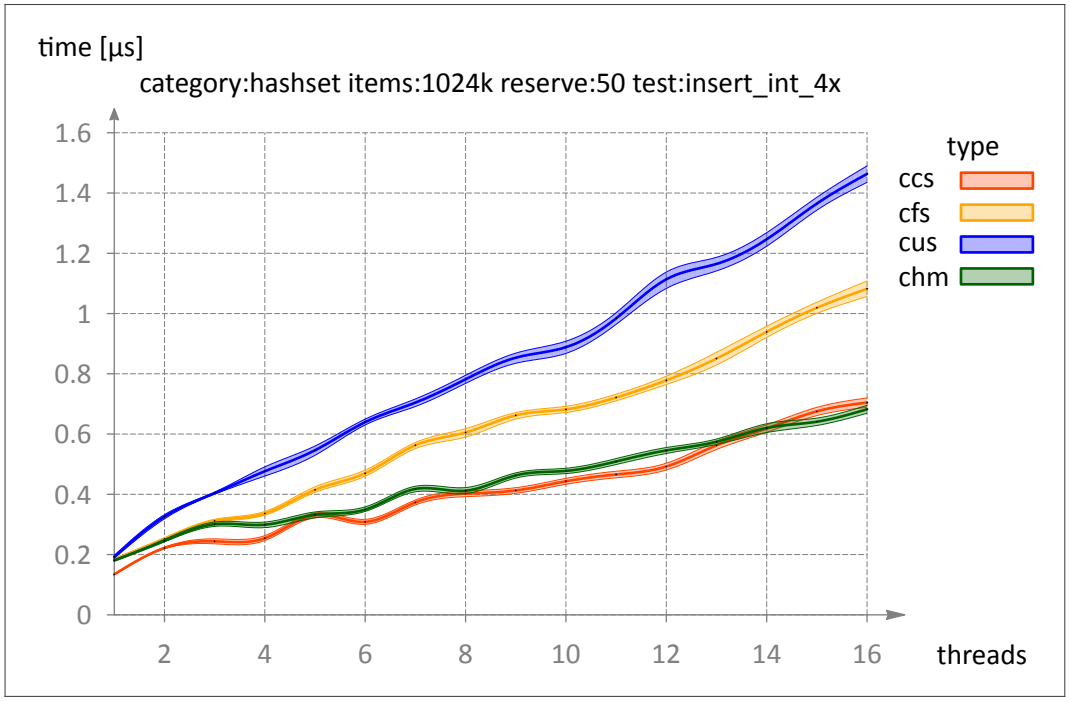
Here we have measured multiple implementations against the number of threads accessing the table at once. As always, all times are scaled both to number of items *and* number of threads. These tests used 1 million and 16 million items respectively. The types of hash tables are following: ccs = "concurrent compact set" (the implementation used in DIVINE), cfs = "concurrent fast set" (less compact, mainly useful with variable-length keys), cus = "concurrent unsorted set" an implementation provided by Intel TBB [96] and chm = "concurrent hash map" (another data structure provided by Intel TBB). We can see that at 1M items, the implementations are about break-even. At 16M items, the more cache-efficient implementations in DIVINE show a pronounced advantage up until the number of physical cores is reached – 12 in this case. The plot for 1M items, 50 % of duplicate keys can be found in **Figure 3.4**.

1M items
insert once



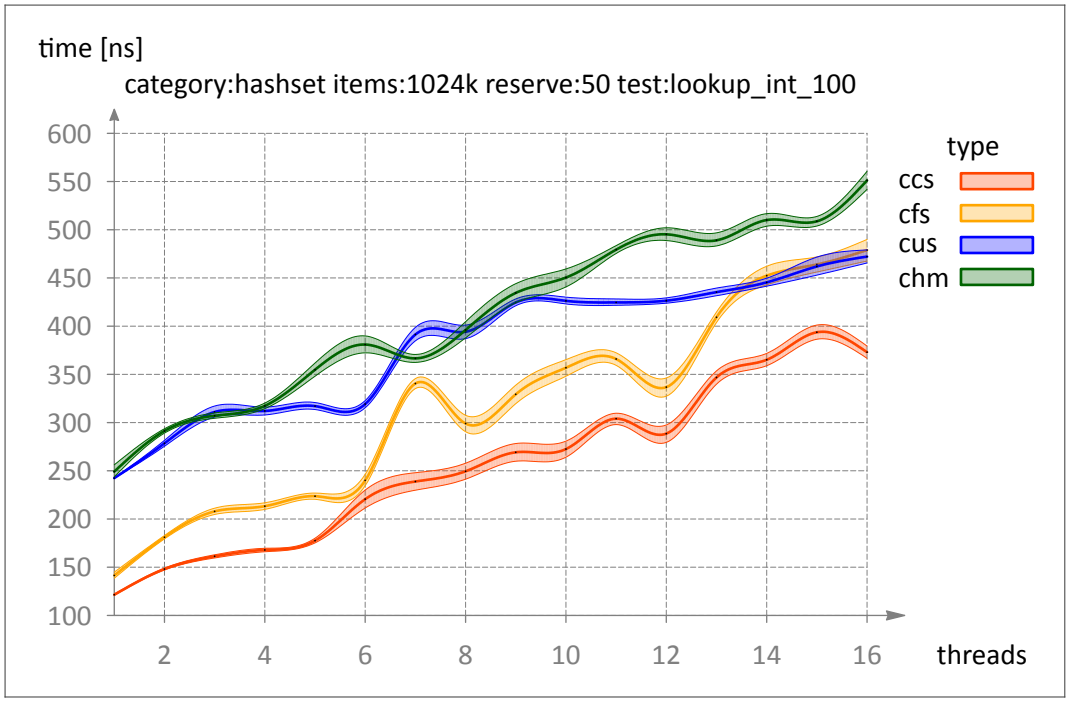
B.8

1M items
insert 4×



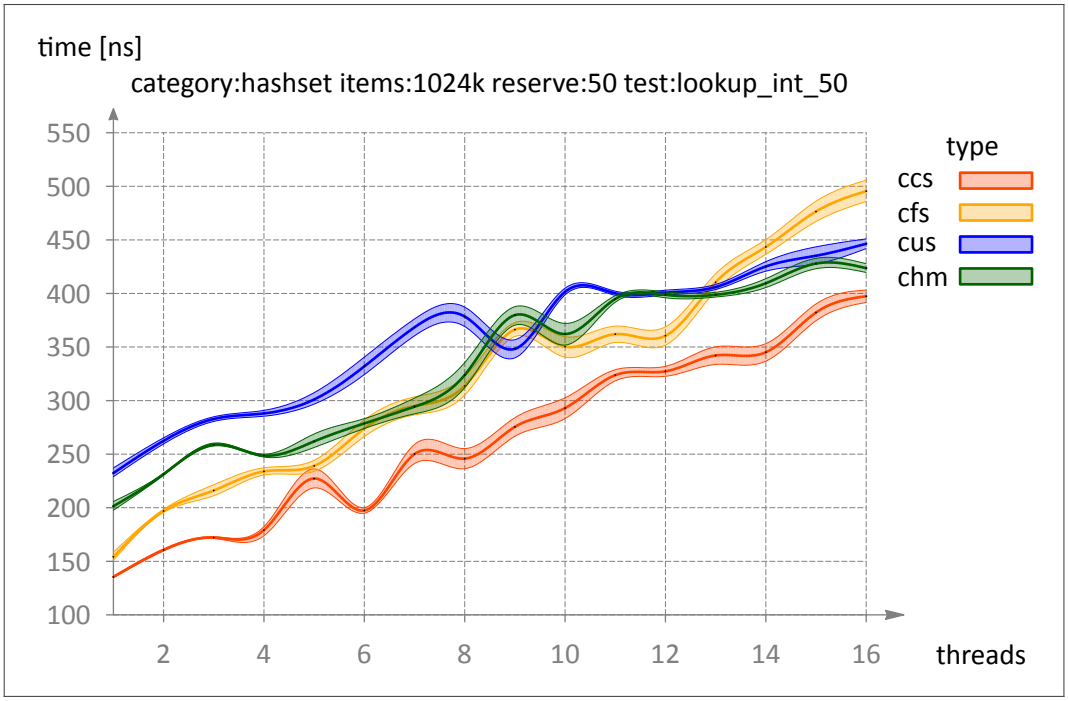
B.9

1M items
lookup, $^{1}/_{1}$ hits



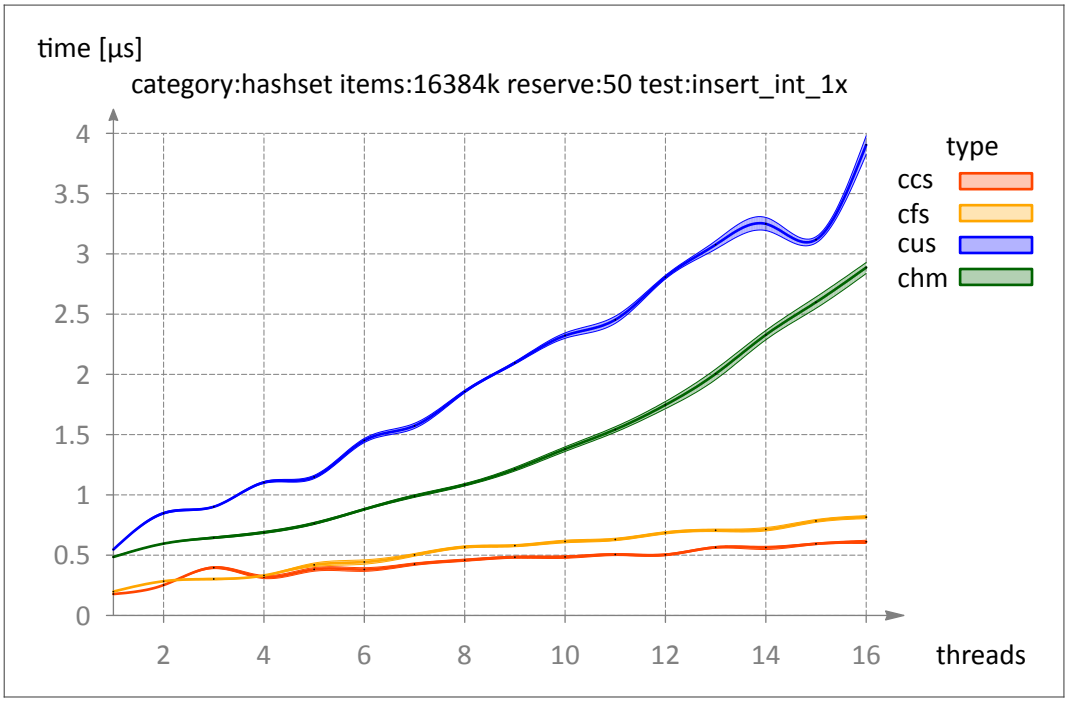
B.10

1M items
lookup, $^1/_2$ hits



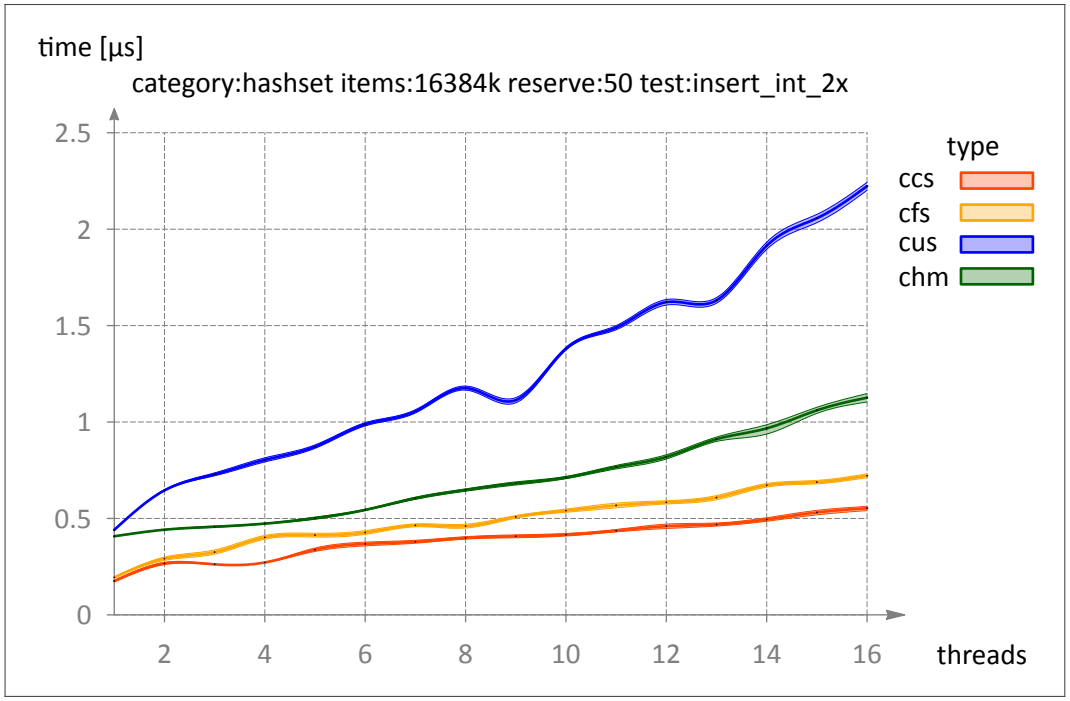
B.11

16M items
insert once



B.12

16M items
lookup, $^1/_2$ hits



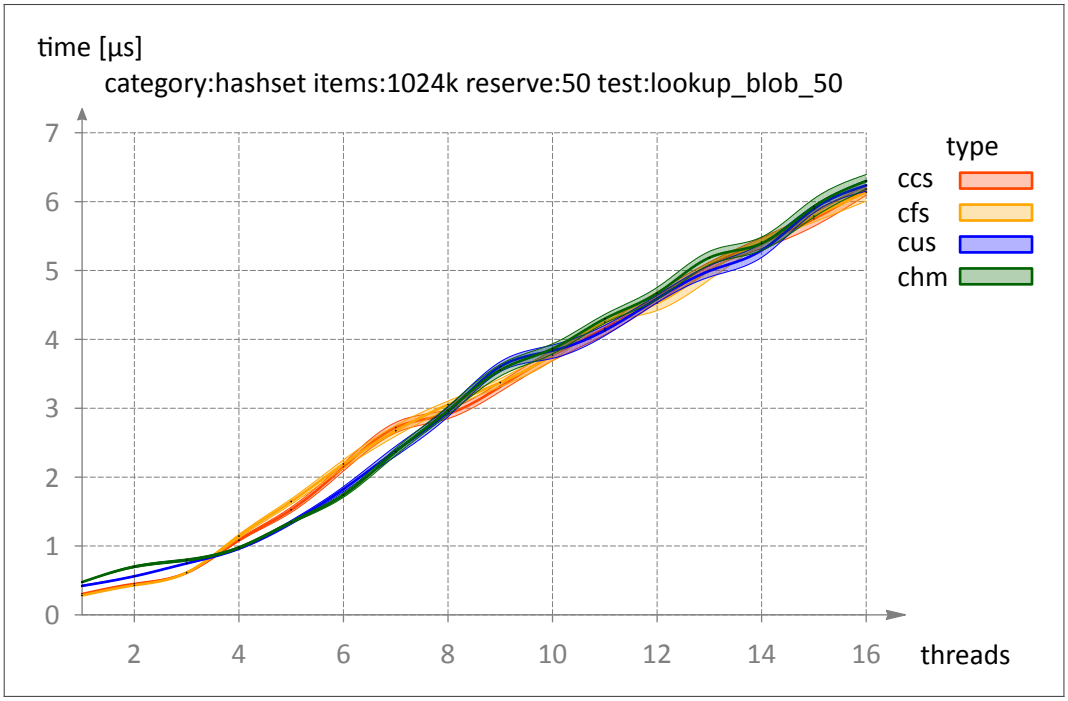
B.13

B.3.3 Scalability of Parallel Access, Long Keys

1M items
insert once

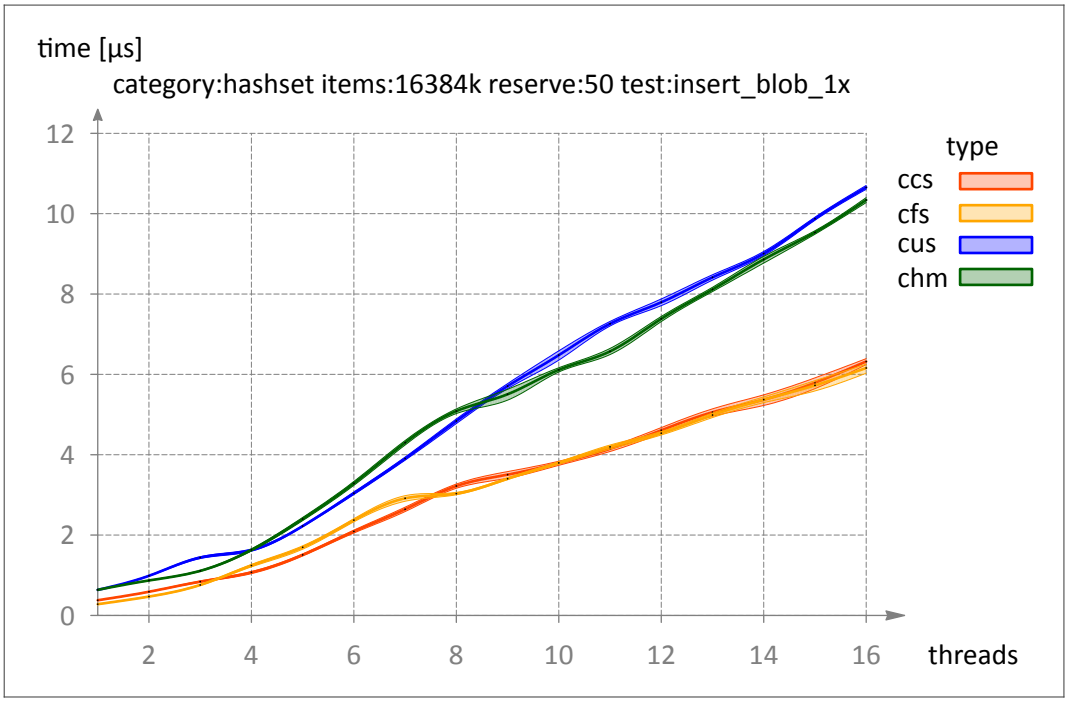
B.14

1M items
lookup, $^1/_2$ hits



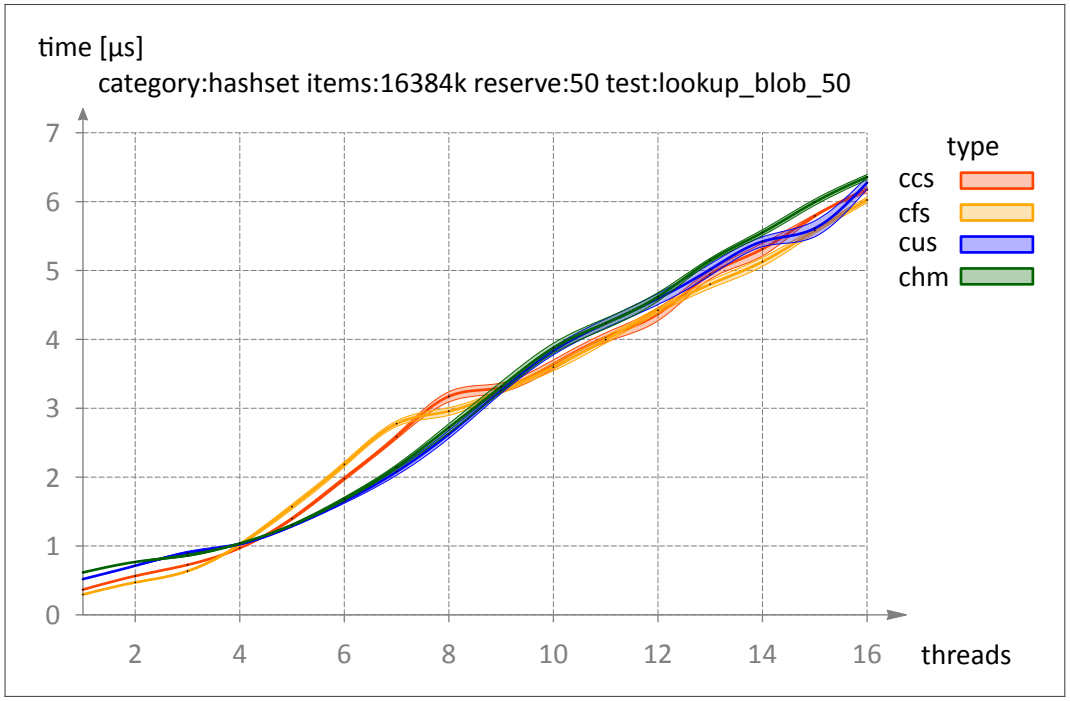
B.15

16M items
insert once



B.16

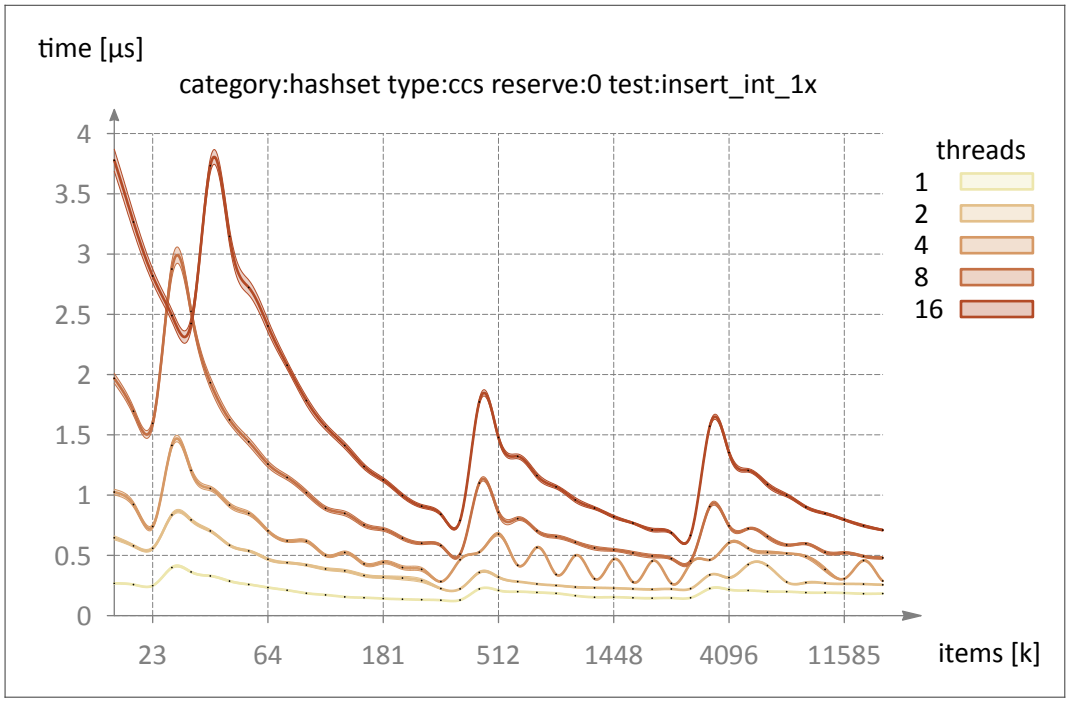
16M items
lookup, $^1/_2$ hits



B.17

B.3.4 Scalability with Size

CCS
insert once



B.18

C Related Papers

The author of this thesis has co-authored the following conference & journal papers. All publication venues are international unless noted otherwise.

2006 *DiVinE – A Tool for Distributed Verification* [7] is a CAV tool paper based on an old implementation of DiVinE targeted at MPI clusters; *contribution:* user interface, execution and supervision of verification jobs in a cluster, copy-editing, overall about 15 %

2007 *Scalable Multi-core LTL Model-Checking* [10], presented at the SPIN workshop on model checking software with focus on adapting distributed-memory algorithms to execute efficiently in shared memory; *contribution:* theoretical background, implementation, benchmarking, writing and copy-editing, overall ca. 50 %

Shared Hash Tables in Parallel Model Checking [22], a paper presented at PDMC, exploring the use of shared hash tables to improve performance of model checking in shared memory; *contribution:* idea, theoretical background, implementation, benchmarking, writing and copy-editing, ca. 66 % in total

2008 *DiVinE Multi-Core – A Parallel LTL Model-Checker* [11], an ATVA tool paper, presenting the first shared-memory version of DIVINE, based on the theoretical and prototyping work described in the 2 previous papers; *contribution:* implementation, release engineering, writing, copy-editing, about 50 %

2009 *An Optimal On-the-fly Parallel Algorithm for Model Checking of Weak LTL Properties* [12], an ICFEM presentation of a new combined algorithm for distributed- and shared-memory accepting cycle detection; *contribution:* collaboration on algorithm design, implementation, benchmarking, writing & copy-editing, ca. 45 % overall

DiVinE 2.0: High-Performance Model Checking [13], a HiBi tool paper on a version of DIVINE with improved performance and the ability to use clusters of multi-core machines efficiently; *contribution:* idea, implementation, release engineering, writing, about 66 % overall

2010 *Parallel Partial Order Reduction with Topological Sort Proviso* [14], a SEFM presentation of a new **C3** check for use with parallel LTL model checking algorithms; *contribution:* idea, algorithm, implementation, benchmarking, proofs, writing & copy-editing, ca. 80 %

Scalable Shared Memory LTL Model Checking [9], a paper published in the International Journal on Software Tools for Technology Transfer, elaborating the techniques for efficient shared memory parallelism in graph exploration, as applied to accepting cycle detection and LTL model checking; *contribution:* theoretical background, writing, implementation, benchmarking, copy-editing, about 50 % overall

DiVinE: Parallel Distributed Model Checker [18], a HiBi/PDMC tool paper on a substantially improved release of DIVINE, with hybrid distributed- and shared-memory capabilities, parallel partial order reduction and other improvements; *contribution:* implementation, release engineering, testing, writing, copy-editing, about 66 % in total

2012 *Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs* [16], a NFM presentation of the new LLVM interpreter for DIVINE; *contribution:* idea, design, implementation, theoretical background, writing, ca. 75 %

On-the-fly Parallel Model Checking Algorithm that is Optimal for Verification of Weak LTL Properties [15], a journal presentation of the algorithm designed in [12] and its combination with parallel partial order reduction for Science of Computer Programming; *contribution:* idea, algorithm design, implementation, benchmarking, theoretical background, writing, overall about 55 %

Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs [6], an FMICS presentation of applying model checking to industrial avionics designs using DIVINE; *contribution:* part of the implementation, copy-editing, theoretical background, about 25 % total

2013 *Improved State Space Reductions for LTL Model Checking of C & C++ Programs* [133], a NFM presentation of the $\tau+$reduction together with heap symmetry reduction and other efficiency improvements in the LLVM interpreter in DIVINE; *contribution:* idea, theoretical background, implementation, writing, ca. 85 %

Distributed LTL Model Checking with Hash Compaction [21], a PDMC presentation that revisited the idea of using hash compaction to reduce memory requirements of LTL model checking in distributed memory using the OWCTY algorithm; *contribution:* idea, writing, correctness proof, part of the implementation, approx. 33 % in total

DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs [8], a CAV tool paper reporting on the LLVM model checking capability of DIVINE and numerous other improvements; *contribution:* theoretical background, writing, implementation, release management, providing guidance to junior co-authors, coordination; ca. 40 % total

2014 *Model Checking C++ with Exceptions* [134], an AVoCS presentation of the extensions to DIVINE's LLVM interpreter required for model checking of C++ programs with exceptions; *contribution:* idea, design, implementation, writing; overall about 85 %

Context-Switch-Directed Verification in DIVINE [154], a presentation at a local workshop (MEMICS) about leveraging directed search for counterexample discovery in LLVM programs using DIVINE; *contribution:* idea, copy-editing, guidance, overall 33 %

References

1 S. Anand, P. Godefroid and N. Tillmann 2008. Demand-Driven Compositional Symbolic Execution. In *TACAS*, p. 367–381.

2 T. Andrews, S. Qadeer, S. Rajamani, J. Rehof and Y. Xie 2004. Zing: A Model Checker for Concurrent Software. In *Computer Aided Verification*, n. 3114 in LNCS, p. 28–32. Springer.

3 A. Armando, J. Mantovani and L. Platania 2006. Bounded model checking of software using SMT solvers instead of SAT solvers. In *13th International Conference on Model Checking Software*, p. 146–162. Springer.

4 T. Ball, B. Cook, V. Levin and S. K. Rajamani 2004. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *Integrated Formal Methods*, p. 1–20. Springer.

5 J. Barnat, P. Bauch and V. Havel 2014. Model Checking Parallel Programs with Inputs. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, p. 756–759. IEEE.

6 J. Barnat, J. Beran, L. Brim, T. Kratochvíla and P. Ročkai 2012. Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs. In *Formal Methods for Industrial Critical Systems (FMICS)*, n. 7437 in LNCS, p. 78–92. Springer.

7 J. Barnat, L. Brim, I. Černá, P. Moravec and P. Ročkai et al. 2006. DiVinE – A Tool for Distributed Verification (Tool Paper). In *Computer Aided Verification*, n. 4144 in LNCS, p. 278–281. Springer.

8 J. Barnat, L. Brim, V. Havel, J. Havlíček and J. Kriho et al. 2013. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification*, n. 8044 in LNCS, p. 863–868. Springer.

9 J. Barnat, L. Brim and P. Ročkai 2010. Scalable Shared Memory LTL Model Checking. In *International Journal on Software Tools for Technology Transfer (STTT)*, 12(2):139–153.

10 J. Barnat, L. Brim and P. Ročkai 2007. Scalable Multi-core LTL Model-Checking. In *Model Checking Software / The International SPIN Workshop*, n. 4595 in LNCS, p. 187–203. Springer.

11 J. Barnat, L. Brim and P. Ročkai 2008. DiVinE Multi-Core – A Parallel LTL Model-Checker. In *Automated Technology for Verification and Analysis*, n. 5311 in LNCS, p. 234–239. Springer.

12 J. Barnat, L. Brim and P. Ročkai 2009. An Optimal On-the-fly Parallel Algorithm for Model Checking of Weak LTL Properties. In *International Conference on Formal Engineering Methods*, n. 5885 in LNCS, p. 407–425. Springer.

13 J. Barnat, L. Brim and P. Ročkai 2009. DiVinE 2.0: High-Performance Model Checking. In *2009 International Workshop on High Performance Computational Systems Biology*, p. 31–32. IEEE Computer Society Press.

14 J. Barnat, L. Brim and P. Ročkai 2010. Parallel Partial Order Reduction with Topological Sort Proviso. In *Software Engineering & Formal Methods*, p. 222–231. IEEE Computer Society Press.
15 J. Barnat, L. Brim and P. Ročkai 2012b. On-the-fly Parallel Model Checking Algorithm that is Optimal for Verification of Weak LTL Properties. In *Science of Computer Programming*, 77(12):1272–1288.
16 J. Barnat, L. Brim and P. Ročkai 2012a. Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs. In *NASA Formal Methods Symposium*, n. 7226 in LNCS, p. 252–267. Springer.
17 J. Barnat, L. Brim and I. Černá 2005. Cluster-Based LTL Model Checking of Large Systems. In *Formal Methods for Components and Objects*, n. 4111 in LNCS, p. 259–279.
18 J. Barnat, L. Brim, M. Češka and P. Ročkai 2010. DiVinE: Parallel Distributed Model Checker (Tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC)*, p. 4–7. IEEE.
19 J. Barnat, L. Brim and P. Šimeček 2007. I/O Efficient Accepting Cycle Detection. In *Computer Aided Verification*, n. 4590 in LNCS, p. 281–293. Springer.
20 J. Barnat, L. Brim and P. Šimeček 2009. Cluster-Based I/O Efficient LTL Model Checking. In *IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, p. 635–639. IEEE Computer Society.
21 J. Barnat, J. Havlíček and P. Ročkai 2013. Distributed LTL Model Checking with Hash Compaction. In *PASM/PDMC 2012*, p. 79-93. ENTCS.
22 J. Barnat and P. Ročkai 2007. Shared Hash Tables in Parallel Model Checking. In *The International Workshop on Parallel and Distributed Methods in verifiCation (PDMC)*, p. 81–95. CTIT, University of Twente.
23 V. Batagelj 1975. The Quadratic Hash Method When the Table Size Is Not a Prime Number. In *Communications of ACM*, 18(4):216–217.
24 J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson and W. Yi 1996. *UPPAAL – A Tool Suite for Automatic Verification of Real-Time Systems*. Springer.
25 E. D. Berger, K. S. McKinley, R. D. Blumofe and P. R. Wilson 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, p. 117–128.
26 A. Biere, C. Artho and V. Schuppan 2002. Liveness Checking as Safety Checking. In *Electronic Notes in Theoretical Computer Science*, 66(2):160–177.
27 A. Biere, A. Cimatti, E. M. Clarke and Y. Zhu 1999. Symbolic Model Checking without BDDs. In *TACAS*, p. 193–207.
28 S. Blom, B. Lisser, J. van de Pol and M. Weber 2008. A Database Approach to Distributed State Space Generation. In *Electronic Notes in Theoretical Computer Science*, 198(1):17–32.
29 H. Boehm 1993. Space Efficient Conservative Garbage Collection.

30 D. Bosnacki, S. Leue and A. L. Lafuente 2009. Partial-Order Reduction for General State Exploring Algorithms. In *International Journal on Software Tools for Technology Transfer*, 11(1):39–51.

31 J. P. Bowen and M. G. Hinchey 1995. Ten Commandments of Formal Methods. In *Computer*, 28(4):56–63.

32 L. Brim, J. Barnat, T. Janoušek, P. Ročkai and V. Štill 2014. DIVINE Manual.

33 L. Brim, I. Černá, P. Moravec and J. Šimša 2004. Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In *The International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, n. 3312 in LNCS, p. 352–366. Springer.

34 L. Brim, I. Černá, P. Moravec and J. Šimša 2005b. Distributed Partial Order Reduction of State Spaces. In *Electronic Notes on Theoretical Compututer Science*, 128(3):63–74.

35 L. Brim, I. Černá, P. Moravec and J. Šimša 2005a. How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors. In *The International Workshop on Parallel and Distributed Methods in verifiCation (PDMC)*, p. 1–12.

36 L. Brim, I. Černá, P. Vařeková and B. Zimmerová 2006. Component-Interaction Automata as a Verification-Oriented Component-Based System Specification. In *ACM SIGSOFT Software Engineering Notes*, 31(2).

37 R. E. Bryant 1991. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. In *IEEE Trans. Computers*, 40(2):205–213.

38 J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill and L. J. Hwang 1992. Symbolic Model Checking: 10^{20} States and Beyond. In *Information and Computation*, 98(2):142–170.

39 C. Cadar, D. Dunbar and D. R. Engler 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, p. 209–224.

40 S. Caselli, G. Conte and P. Marenzoni 1995a. Parallel State Space Exploration for GSPN Models. In *Application and Theory of Petri Nets*, p. 181–200.

41 S. Caselli, G. Conte and P. Marenzoni 1995b. Parallel State Space Exploration for GSPN models. In *Applications and Theory of Petri Nets 1995*, n. 935 in LNCS, p. 181–200. Springe.

42 I. Černá and R. Pelánek 2003. Distributed Explicit Fair Cycle Detection (Set Based Approach). In *Model Checking Software / The International SPIN Workshop*, n. 2648 in LNCS, p. 49–73. Springer.

43 G. Ciardo, J. Gluckman and D. M. Nicol 1998. Distributed State Space Generation of Discrete-State Stochastic Models. In *INFORMS Journal on Computing*, 10(1):82–93.

44 K. Claessen and J. Hughes 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, p. 268–279. ACM.

45 E. M. Clarke, A. Biere, R. Raimi and Y. Zhu 2001. Bounded Model Checking Using Satisfiability Solving. In *Formal Methods in System Design*, 19(1):7–34.

46 E. M. Clarke and E. A. Emerson 1981. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs*, p. 52–71.

47 E. M. Clarke, E. A. Emerson, S. Jha and A. P. Sistla 1998. Symmetry Reductions in Model Checking. In *Computer Aided Verification*, p. 147–158.

48 E. M. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith 2000. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, n. 1855 in LNCS, p. 154–169. Springer.

49 E. M. Clarke, O. Grumberg and D. E. Long 1994. Model Checking and Abstraction. In *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542.

50 E. M. Clarke, O. Grumberg and D. Peled 1999. *Model Checking*. MIT press.

51 E. M. Clarke, D. Kroening and F. Lerda 2004. A Tool for Checking ANSI-C Programs. In *TACAS*, p. 168–176.

52 L. A. Clarke 1976. A Program Testing System. In *The 1976 Annual Conference*, p. 488–491. ACM.

53 DWARF Debugging Information Format Committee 2010. *DWARF debugging information format version 4*.

54 J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach and C. S. Pasareanu et al. 2000. Bandera: Extracting Finite-state Models from Java Source Code. In *International Conference on Software Engineering*, 0:439.

55 C. Courcoubetis, M. Y. Vardi, P. Wolper and M. Yannakakis 1992. Memory-Efficient Algorithms for the Verification of Temporal Properties. In *Formal Methods in System Design*, 1:275–288.

56 D. L. Dill 1996. The Murphi Verification System. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, n. 1102 in LNCS, p. 390–393. Springer.

57 N. A. de Brugh, V. Nguyen and T. Ruys 2009. MoonWalker: Verification of .NET Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, n. 5505 in LNCS, p. 170–173. Springer.

58 C. DeMartini, R. Iosif and R. Sisto 1999. A Deadlock Detection Tool for Concurrent Java Programs. In *Software Practice and Experience*, 29(7):577–603.

59 E. W. Dijkstra 1987. Shmuel Safra's Version of Termination Detection. EWD Manuscript. http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD998.PDF

60 S. Edelkamp and S. Jabbar 2006. Large-scale directed model checking LTL. In *SPIN'06*, p. 1–18. Springer.

61 S. Edelkamp, S. Leue and A. Lluch-Lafuente 2004. Directed explicit-state model checking in the validation of communication protocols. In *International Journal on Software Tools for Technology Transfer*, 5(2-3):247–267.

62 S. Edelkamp, A. Lluch-Lafuente and S. Leue 2001. Directed Explicit Model Checking with HSF-SPIN. In *SPIN'01*, p. 57–79. Springer.

63 S. Edelkamp, V. Schuppan, D. Bosnacki, A. Wijs and A. Fehnker et al. 2008. Survey on Directed Model Checking. In *MoChArt*, p. 65–89.

64 E. A. Emerson 1990. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science*, p. 995–1072. Elsevier.

65 E. A. Emerson and J. Y. Halpern 1986. ``Sometimes'' and ``Not Never'' Revisited: on Branching versus Linear Time Temporal Logic. In *J. ACM*, 33(1):151–178.

66 E. A. Emerson and A. P. Sistla 1996. Symmetry and Model Checking. In *Formal Methods in System Design*, 9(1-2):105–131.

67 K. Esseghir 1993. Improving Data Locality for Caches. Master's thesis, Department of Computer Science, Rice University.

68 S. Evangelista, A. W. Laarman, L. Petrucci and J. C. van de Pol 2012. Improved Multi-Core Nested Depth-First Search. In *International Symposium on Automated Technology for Verification and Analysis, ATVA 2012*, n. 7561 in LNCS, p. 269–283. Springer.

69 S. Evangelista, L. Petrucci and S. Youcef 2011. Parallel Nested Depth-First Searches for LTL Model Checking. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis*, n. 6996 in LNCS, p. 381–396. Springer.

70 R. Fagin, J. Nievergelt, N. Pippenger and H. R. Strong 1979. Extendible Hashing -— A Fast Access Method for Dynamic Files. In *ACM Transactions on Database Systems*, 4(3):315–344.

71 S. Falke, F. Merz and C. Sinz 2013. The bounded model checker LLBMC. In *ASE*, p. 706–709. IEEE.

72 K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi and Z. Yang 2001a. Is there a best symbolic cycle-detection algorithm?. In *Tools and Algorithms for the Construction and Analysis of Systems*, n. 2031 in LNCS, p. 420–434. Springer.

73 K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi and Z. Yang 2001b. Is There a Best Symbolic Cycle-Detection Algorithm?. In *Tools and Algorithms for the Construction and Analysis of Systems*, p. 420–434. Springer.

74 C. Flanagan and S. Qadeer 2003. Transactions for Software Model Checking. In *The Workshop on Software Model Checking*, p. 338–349. Elsevier.

75 H. Garavel, R. Mateescu and I. Smarandache 2001. Parallel State Space Construction for Model-Checking. In *Model Checking Software*, n. 2057 in LNCS, p. 217–234. Springer.

76 J. Geldenhuys, P. de Villiers and J. Rushby 1999. Runtime Efficient State Compaction in SPIN. In *Theoretical and Practical Aspects of SPIN Model Checking*, n. 1680 in LNCS, p. 12–21. Springer.

77 P. Godefroid 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032. Springer.

78 P. Godefroid and D. Pirottin 1993. Refining Dependencies Improves Partial-Order Verification Methods. In *Computer Aided Verification*, p. 438–449.

79 H. Günther and G. Weissenbacher 2014. Incremental Bounded Software Model Checking. In *SPIN*. ACM.

80 T. Harris, S. Marlow, S. P. Jones and M. Herlihy 2005. Composable Memory Transactions. In *An ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

81 T. Henzinger, R. Jhala, R. Majumdar and G. Sutre 2003. Software Verification with BLAST. In *Model Checking Software*, n. 2648 in LNCS, p. 624–624. Springer.

82 M. Herlihy and J. B. E. Moss 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *SIGARCH Compututer Architecture News*, 21:289–300.

83 D. W. Hillis and G. L. Steele Jr. 1986. Data Parallel Algorithms. In *Communications of ACM*, 29:1170–1183.

84 G. Holzmann 2000. Logic Verification of ANSI-C Code with SPIN. In *SPIN Model Checking and Software Verification*, n. 1885 in LNCS, p. 131–147. Springer.

85 G. J. Holzmann 1988. An Improved Protocol Reachability Analysis Technique. In *Software, Practice and Experience*, 18:137–161.

86 G. J. Holzmann 1997. State Compression in SPIN: Recursive Indexing And Compression Training Runs. In *The International SPIN Workshop*.

87 G. J. Holzmann 2003. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley.

88 G. J. Holzmann 2007. A Stack-Slicing Algorithm for Multi-Core Model Checking. In *Workshop on Parallel and Distributed Methods in verifiCation (PDMC)*, p. 1–15. CTIT, University of Twente.

89 G. J. Holzmann and D. Bosnacki 2007. The Design of a Multicore Extension of the SPIN Model Checker. In *Software Engineering, IEEE Transactions on*, 33:659–674.

90 G. J. Holzmann, P. Godefroid and D. Pirottin 1992. Coverage Preserving Reduction Strategies for Reachability Analysis. In *PSTV*, p. 349–363.

91 G. J. Holzmann, R. Joshi and A. Groce 2011. Swarm Verification Techniques. In *IEEE Transactions on Software Engineering*, 37(6):845–857.

92 G. J. Holzmann, D. Peled. and M. Yannakakis 1996. On Nested Depth First Search. In *The SPIN Verification System*, p. 23–32. American Mathematical Society. Proc. of the 2nd SPIN Workshop.

93 G. J. Holzmann and M. H. Smith 2001. Software Model Checking: Extracting Verification Models from Source Code. In *Software Testing, Verification and Reliability*, 11(2):65–79.

94 IEEE 1995. *Std 1003.1c-1995 Standard for Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. IEEE Computer Society Press.

95 IEEE 2001. *Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. IEEE Computer Society Press. (Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6).

96 Intel Corporation 2014. Threading Building Blocks (2014-06-01). http://www.threadingbuildingblocks.org

97 R. J. Jenkins 1997. Hash Functions. In *Dr Dobbs Journal*, 22(9):107–+.

98 R. J. Jenkins 2006. A Hash Function for Hash Table Lookup. http://burtleburtle.net/bob/hash/doobs.html

99 R. J. Jenkins 2012. SpookyHash: a 128-bit Noncryptographic Hash. http://burtleburtle.net/bob/hash/spooky.html

100 R. Jhala and R. Majumdar 2005. Path Slicing. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, p. 38–47. ACM Press.

101 R. Jones and R. D. Lins 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons.

102 R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer and J. Whittemore et al. 2009. Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation. In *Computer Aided Verification*, n. 5643 in LNCS, p. 414–429. Springer.

103 D. Kozen 1983. Results on the Propositional μ-calculus. In *Theoretical Computer Science*, 27(3):333–354. Special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982.

104 S. A. Kripke 1963. Semantical Considerations on Modal Logic. In *Acta Philosophica Fennica*, 16:83–94.

105 D. Kroening and M. Tautschnig 2014. CBMC – C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, n. 8413 in LNCS, p. 389–391. Springer.

106 O. Krč-Jediný 2010. GIMPLE Model Checker. Master's thesis, Faculty of Mathematics and Physics, Charles University in Prague.

107 R. P. Kurshan, V. Levin, M. Minea, D. Peled and H. Yenigün 1998. Static Partial Order Reduction. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, n. 1384 in LNCS, p. 345–357. Springer.

108 A. Laarman, R. Langerak, J. van de Pol, M. Weber and A. Wijs 2011. Multi-core Nested Depth-First Search. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis*, n. 6996 in LNCS, p. 321–335. Springer Berlin Heidelberg.

109 A. Laarman and J. van de Pol 2011. Variations on Multi-Core Nested Depth-First Search. In *ArXiv e-prints*.

110 A. Laarman, J. van de Pol and M. Weber 2011. Parallel Recursive State Compression for Free. In *SPIN*, p. 38–56.

111 A. W. Laarman 2014. *Scalable Multi-Core Model Checking*. PhD thesis, http://fmt.cs.utwente.nl/tools/ltsmin/laarman_thesis/

112 Leslie Lamport 1994. The Temporal Logic of Actions. In *ACM Trans. Program. Lang. Syst.*, 16(3):872–923.

113 C. Lattner and V. Adve 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*.

114 X. Leroy 2009. Formal verification of a realistic compiler. In *Communications of the ACM*, 52(7):107–115.

115 LLVM Project 2014. LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html
116 K. K. Ma, K. Y. Phang, J. S. Foster and M. Hicks 2011. Directed Symbolic Execution. In *International Conference on Static Analysis (SAS)*, p. 95–111. Springer.
117 K. L. McMillan 2006. Lazy Abstraction with Interpolants. In *Computer Aided Verification*, p. 123–136. Springer.
118 Message Passing Interface Forum 2008. MPI: A Message-Passing Interface Standard Version 2.1. http://www.mpi-forum.org/docs/mpi21-report.pdf
119 M. M. Michael 2004. Scalable Lock-Free Dynamic Memory Allocation. In *SIGPLAN Notices*, 39(6):35–46.
120 M. M. Michael and M. L. Scott 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Symposium on Principles of Distributed Computing*, p. 267–275.
121 P. Moravec 2008. *Distributed State Space Reductions*. PhD thesis, Faculty of Informatics, Masaryk University Brno.
122 M. S. Musuvathi, D. Park, A. Chou, D. R. Engler and D. L. Dill 2002. CMC: A Pragmatic Approach to Model Checking Real Code. In *The Fifth Symposium on Operating Systems Design and Implementation*.
123 N. Nethercote and J. Seward 2007. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, p. 89–100. ACM.
124 R. Palmer and G. Gopalakrishnan 2002. Partial Order Reduction Assisted Parallel Model Checking. In *Parallel and Distributed Model Checking (PDMC) Workshop*.
125 R. Pelánek 2007. BEEM: Benchmarks for Explicit Model Checkers. In *Model Checking Software / The International SPIN Workshop*, n. 4595 in LNCS, p. 263–267. Springer.
126 R. Pelánek 2008. Fighting State Space Explosion: Review and Evaluation. In *Formal Methods for Industrial Critical Systems (FMICS)*, p. 47–62. ERCIM.
127 D. Peled 1993. All from One, One for All: on Model Checking Using Representatives. In *Computer Aided Verification*, p. 409–423. Springer.
128 D. Peled 1994. Combining Partial Order Reductions with On-the-fly Model-Checking. In *Computer Aided Verification*, p. 377–390. Springer.
129 D. Peled 1998. Ten Years of Partial Order Reduction. In *Computer Aided Verification*, p. 17–28. Springer.
130 G. Pike and J. Alakuijala 2011. Introducing CityHash.
131 A. Rabinovich 2014. A Proof of Kamp's Theorem. In *Logical Methods in Computer Science*, 10(1).
132 J. H. Reif 1985. Depth-First Search is Inherently Sequential. In *Information Processing Letters*, 20(5):229–234.
133 P. Ročkai, J. Barnat and L. Brim 2013. Improved State Space Reductions for LTL Model Checking of C & C++ Programs. In *NASA Formal Methods Symposium*, n. 7871 in LNCS, p. 1–15. Springer.

134 P. Ročkai, J. Barnat and L. Brim 2014. Model Checking C++ with Exceptions. In *To appear in Electronic Communications of the EASST, Proceedings of Automated Verification of Critical Systems*.

135 C. Runciman, M. Naylor and F. Lindblad 2008. SmallCheck and Lazy SmallCheck: Automatic Exhaustive Testing for Small Values. In *Haskell*, p. 37–48.

136 O. Saarikivi, K. Kahkonen and K. Heljanko 2012. Improving Dynamic Partial Order Reductions for Concolic Testing. In *Application of Concurrency to System Design (ACSD), 2012 12th International Conference on*, p. 132–141.

137 J. R. Scientist 1984. Citation Needed. Nonlinear Publishing.

138 K. Sen, D. Marinov and G. Agha 2005. CUTE: A Concolic Unit Testing Engine for C. In *European Software Engineering Conference (ESEC/FSE)*, p. 263–272. ACM.

139 N. Shavit and D. Touitou 1995. Software Transactional Memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, p. 204–213.

140 P. Šimeček 2006. DiVinE – Distributed Verification Environment. Master's thesis, Faculty of Informatics, Masaryk University Brno.

141 H. Sivaraj and G. Gopalakrishnan 2003. Random Walk Based Heuristic Algorithms for Distributed Memory Model Checking. In *Electronic Notes in Theoretical Computer Science*, 89(1):51–67.

142 M. Staats and C. Păsăreanu 2010. Parallel Symbolic Execution for Structural Test Generation. In *International Symposium on Software Testing and Analysis (ISSTA)*, p. 183–194. ACM.

143 U. Stern and D. L. Dill 1995. Improved Probabilistic Verification by Hash Compaction. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, p. 206–224. Springer.

144 S. A. M. Talbot 1995. Performance Tuning of Programs for Shared-Memory Multiprocessors. Master's thesis, Imperial College, London, UK.

145 R. Tarjan 1972. Depth First Search and Linear Graph Algorithms. In *SIAM Journal on Computing*, p. 146–160.

146 A. Tarski 1955. A Lattice-Theoretical Fixpoint Theorem and its Applications.

147 J. Torrellas, M. S. Lam and J. L. Hennessy 1994. False Sharing and Spatial Locality in Multiprocessor Caches. In *Computers, IEEE Transactions on*, 43(6):651–663.

148 D. Turner 2001. Robust Design Techniques for C Programs. http://freetype.sourceforge.net/david/reliable-c.html

149 A. Valmari 1997. Stubborn Set Methods for Process Algebras. In *DIMACS workshop on Partial Order Methods in Verification*, p. 213–231. AMS Press, Inc..

150 M. Y. Vardi and P. Wolper 1986. An Automata-Theoretic Approach to Automatic Program Verification. In *IEEE Symposium on Logic in Computer Science*, p. 322–331. Computer Society Press.

151 W. Visser, K. Havelund, G. P. Brat and S. Park 2000. Model Checking Programs. In *ASE*, p. 3–12.

152 O. Šerý 2010. *Automated Verification of Software*. PhD thesis, Faculty of Mathematics and Physics, Charles University in Prague.

153 V. Štill 2013. State Space Compression for the DiVinE Model Checker. Bachelor's thesis, Faculty of Informatics, Masaryk University Brno.
154 V. Štill, P. Ročkai and J. Barnat 2014. Context-Switch-Directed Verification in DIVINE. In *Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*.
155 J. Weiser 2013. Dynamicky rostoucí sdílená hašovací tabulka pro DiVinE. Bachelor's thesis, Faculty of Informatics, Masaryk University Brno.
156 P. Wolper and D. Leroy 1993. Reliable Hashing without Collision Detection. In *Computer Aided Verification*, p. 59–70. Springer.
157 K. Yorav and O. Grumberg 2004. Static Analysis for State-Space Reductions Preserving Temporal Logics. In *Formal Methods in System Design*, 25(1):67–96.
158 A. Zaks and R. Joshi 2008. Verifying Multi-threaded C Programs with SPIN. In K. Havelund, R. Majumdar and J. Palsberg, editors, *Model Checking Software*, n. 5156 in LNCS, chapter 22, p. 325–342. Springer.
159 J. Zhao, S. Nagarakatte, M. M. K. Martin. and S. Zdancewic 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, p. 427–440. ACM.
160 J. Ziv and A. Lempel 1977. A Universal Algorithm for Sequential Data Compression. In *Information Theory, IEEE Transactions on*, 23(3):337–343.