Automated Concurrency Bug Finding using Partial-Orders

Alex Horn

Balliol College

University of Oxford

Dissertation for the degree of *Doctor of Philosophy*

Trinity 2015

Acknowledgements

Daniel Kroening and Tom Melham, under whose supervision I have learned so much over the last few years, have been a consistent source of inspiration, and I thank them enormously. Through their unwavering support, steady guidance, and the freedom they allowed me to investigate problems from various angles, I have grown to be the researcher and person I am now. Thank you!

There are numerous other people I should like to thank. I am especially grateful to Alan Hu, Luke Ong, Moshe Vardi, and Sharad Malik, who offered illuminating feedback and ideas at our Intel Firmware Verification project meetings. I am also indebted to Michael Tautschnig and Celina Val for their help with the experimental evaluation. I would like to thank Ofer Strichman for the warm welcome we received in Haifa and the discussions we have had in the group there. I thank Jade Alglave for her comments and the opportunity she has created to meet Sir Tony Hoare and John Wickerson, whose feedback was invaluable. I am grateful for Vincent Nimal's friendship, advice and many helpful suggestions on an earlier draft of this thesis. I also would like to express my special gratitude to Liana Hadarean and Tim King, who have enormously enriched my understanding of conflict-driven backtracking solvers. I would like to deeply thank Vojtěch Forejt for his resolute support in so many ways. My sincere thanks also goes to Jim Grundy whose approach to research in industry, mentoring and leadership have inspired me much during my internship at Intel. I also would like to thank Amit Goel and Sava Krstić for their useful suggestions while I was at Intel. Similarly, I would like to thank Nuno Lopes, Andrey Rybalchenko, Nikolaj Bjørner, and Christoph Wintersteiger who made the internship at MSR such a valuable experience. It is with much gratitude that I thank my examiners, Marta Kwiatkowska and Viktor Vafeiadis, for their support and encouragement.

Daniel Kroening brings together a vibrant group of brilliant people from around the globe, and it was a pleasure to meet members of the group

(past and present). I particularly would like to thank Martin Brain, Leopold Haller, and Vijay D'Silva for sharing their insights into abstract interpretation; Dario Cattaruzza and Lihao Liang for their friendship and the kindness they have shown me; César Rodríguez and Marcelo Sousa for the lessons on partial-order reduction; Mehrnoosh Sadrzadeh for sparking my interest in algebra; Ruben Martins, Cristina David, Matt Lewis, Peter Schrammel, and Björn Wachter for fun times around board games and many interesting discussions. I also would like to thank Daniel Neville and Aza Ballard-Whyte for their support in the library as well as Julie Sheppard, Pamela Farries and Elizabeth Polgreen who assisted me with various administrative tasks at the University.

The last few years at Oxford have shaped me tremendously, and I would like to thank Grete Bauder, Nate Harper, James Patrick, Nathan Rose, and Roland Slade for their friendship and their perspectives during this time. I thank Ryan Kavanagh for the conversations on our explorations of the surrounding countryside at Cambridge, and his hospitality at CMU in Pittsburgh. Thank you, Matthew Key, for being such a caring brother amidst a busy life.

Long ago, I met individuals who have planted, in one way or another, important seeds in my life, which has helped me to reflect and stay on course while at Oxford. I particularly would like to thank Bill Chaffin, Drew Clippard, David Edwards, Claire Hendricks, Dave, Susan and David McConeghey, Anna-Maria Müller, Dan Ogden, Ashley Ramsey, Micah Whitacre, Taryn Trousdale, and Lon Wiksell, as well as my teachers from many years ago, Eberhard Heise, Jon Beck, Alan Garvey, and Susan Scheurer, who nurtured my interests and taught me fundamental skills I use regularly. I gratefully acknowledge Intel, Corporation and the Morgan Deters travel fund for providing financial support.

I dedicate this thesis to my family, from the youngest to the eldest, who have been there for me and love me.

Abstract

Concurrent systems are ubiquitous, ranging from multi-core processors to large-scale distributed systems. Yet, the verification of concurrent systems remains a daunting task, and technological advances such as weak memory architectures greatly compound this problem. Such challenges have renewed interest in symbolic encodings of partial-order semantics of concurrency using propositional logic or decidable fragments of first-order logic. The impetus behind these partial-order encodings is the efficiency of evermore highly optimized decision procedures, such as *Boolean satisfiability* (*SAT*) and *Satisfiability Modulo Theory* (*SMT*) solvers.

While methods to effectively use SAT/SMT solvers for automatically finding bugs in sequential software through symbolic techniques, such as bounded model checking or symbolic execution, are well known, this thesis gives theoretical and experimental results that highlight a new set of challenges faced by partial-order encoding techniques. Furthermore, it shows how different solutions to these problems emerge if partial-order encodings are restructured into three separate theory-specific parts. This separation opens up a fresh and fruitful algorithmic perspective on partial-order encodings, including a new partial-order encoding that can drastically reduce the run-time of the analysis prior to calling the SAT/SMT solver, and new insights into the exponential worst-case time complexity of partial-order encodings during SAT/SMT solving. These results are significant as they propose a paradigm shift in how to harness the computational power of SAT/SMT solvers for partial-order encodings.

Publications

Part of the material in this dissertation has been published in the following papers:

1. *Formal Co-Validation of Low-Level Hardware/Software Interfaces*, authored in collaboration with Michael Tautschnig, Celina Val, Liaho Liang, Tom Melham, Jim Grundy and Daniel Kroening, appeared in the proceedings of the conference on Formal Methods in Computer-Aided Design (FMCAD 2013), and is included in parts in Chapter 2.
2. *On Partial Order Semantics for SAT/SMT-Based Symbolic Encodings of Weak Memory Concurrency*, authored in collaboration with Daniel Kroening, appeared in the proceedings of the conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2015), and includes parts of Chapter 3 and 4.
3. *A Concurrency Problem with Exponential DPLL(T) Proofs*, authored in collaboration with Liana Hadarean and Tim King, was peer reviewed by three reviewers at the workshop on Satisfiability Modulo Theories (SMT 2015), and is included in Chapter 4 and 5.
4. *Faster Linearizability Checking via P-Compositionality*, authored in collaboration with Daniel Kroening, appeared in the proceedings of the conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2015), and includes the bulk of the material in Chapter 6.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Concurrent systems are ubiquitous, ranging from multi-core processors to large-scale distributed systems. Yet, the verification of concurrent systems remains a daunting task. The difficulty of the problem primarily stems from at least two possible sources of non-determinism at the same time, namely non-deterministic data inputs and concurrency. This makes the finding of concurrency-related bugs notoriously difficult in practice and, in general, undecidable. Similar to sequential programs, the undecidability of the problem necessitates imposing restrictions on the problem, typically by limiting the class of programs that can be analyzed, or by relaxing the guarantees the analysis makes about the reported presence and absence of bugs.

The work in this thesis investigates concurrency bug finding techniques that are automatic and precise, guided by the research question on how known algorithmic techniques used by non-chronological backtracking decision procedures for NP-hard problems, such as the archetypal Boolean satisfiability problem, could be exploited to more efficiently find concurrency-related bugs. For the purposes of this investigation, an analysis is precise if it generates neither false alarms nor false claims of program correctness. All reported bugs can therefore be exhibited in the concrete semantics of the concurrent system. Perhaps surprisingly, however, what these concurrency semantics exactly should be is a somewhat contentious question. In the past, 'interleaving semantics' (also known as sequential consistency [Lam79a]) would have been adequate.[1] Nowadays, however, the majority of all modern multi-processor architectures can violate interleaving semantics. These violations of interleaving semantics are largely due to technological advances in computer architectures and compilers that aggressively optimize memory accesses, including optimizations such as branch pre-

[1] In interleaving semantics, memory loads and stores in each thread are interleaved such that they form a sequence in which they take effect.

diction, out-of-order execution and CPU-local buffers. This has fueled active research into a new form of shared memory concurrency semantics, collectively known as *weak memory semantics*, whose purpose is to describe the behaviour of shared memory systems under all commonly accepted computer architecture and compiler optimizations, while disallowing too much degree of freedom that can result in undesirable behaviour, such as "out-of-thin-air reads", e.g. [BOS+11, AMT14, PPS16, JR16]. This has proved to be a difficult problem, and so far no consensus has been reached yet on how to solve it for the general case.

For the *algorithmic study of concurrency bug finding techniques* in this thesis, therefore, I make a pragmatic decision and focus only on one of the following agreed upon key characteristics of the problem: memory stores and loads can be reordered.[2] In this family of weak memory semantics, this thesis concerns automated concurrency bug finding techniques that are precise when the effects of such reorderings can be formalized through a partial-order on memory accesses. While finding concurrency bugs is conceptually possible by exhaustively analyzing all linearizations of such a partial-order, this is generally intractable. Next, I recall two prominent methods, partial-order reduction and symbolic techniques, for tackling this problem, and explain why I focus on the latter.

Partial-order reduction (POR) aims to explore a representative subset of all linearizations by partitioning linearizations into the same equivalence class if they reach the same state in the concurrent system. To achieve this form of partitioning, most POR techniques exploit the commutativity, or independence, of concurrently executed transitions [God91, Val92, GW93, Pel93, GW94, Pel96, God96, FG05, YWY06, GFYS07, YCGK08, AAJS14, AAA+15, RSSK15], where each induced equivalence class of linearizations corresponds to a so-called Mazurkiewicz trace [Maz87]. Each Mazurkiewicz trace induced by an independence relation can equivalently be seen as a certain labelled partial-order, traditionally called the dependence graph. Note that Lamport's *happens-before relation* [Lam78], whose practical relevance for distributed systems has long been recognized in the form of vector clocks [Fid88, Mat89], can also be seen as a form of a Mazurkiewicz trace [FG05]. Most recently, optimal POR techniques have been devised [AAJS14, RSSK15] that compute the independence relation dynamically yet explore exactly one representative linearization per Mazurkiewicz trace. However, despite research efforts over the last decade [ABH+01, LST03, CKS05, KGS06, KWG09, WKO13], one of the main challenges of POR re-

[2]The reordering of memory accesses is also known as *de*ordering [PPS16]. Appendix A illustrates a possible reordering of memory accesses and its computational effect.

mains state-space explosion caused by non-deterministic data inputs, a different form of non-determinism than concurrency.

This has motivated purely symbolic techniques that generalize bounded model checking [BCM+92, McM93] and symbolic execution [Kin76] from sequential to concurrent programs. The goal of this generalization is to automatically reason about non-deterministic data inputs and concurrency at the same time. Towards this goal, these techniques symbolically encode the behaviour of the concurrent program into decidable logics. Crucially, finding concurrency-related bugs then reduces to checking whether the generated logic formula is satisfiable. In the case of propositional logic or decidable fragments of first-order logic, these techniques can leverage highly optimized decision procedures, such as *Boolean satisfiability* (*SAT*) and *Satisfiability Modulo Theory* (*SMT*) solvers. What makes this approach so interesting from an algorithmic point of view is that SAT/SMT solvers can automatically 'learn' facts about a given *NP*-hard problem by analyzing each logical conflict encountered during the solving of the formula. This conflict-driven learning can prune large parts of the search space. While POR is also a pruning strategy, it is rather different from conflict-driven learning in SAT/SMT solvers. One of the main differences is that SAT/SMT decision procedures can handle case splits in a uniform way through logical disjunctions, whereas POR currently offers no such generality. The uniform treatment of disjunctions in SAT/SMT solvers poses the research question of how conflict-driven learning in SAT/SMT solvers interacts with symbolic techniques that automatically find concurrency-related bugs.

In the family of symbolic concurrency-related bug finding techniques that exploit SAT/SMT solvers, sequentialization techniques translate a concurrent system into a sequential program under a bounded number of context switches (e.g. [ITF+14]) or memory writes (e.g. [TIF+15]). To achieve this translation, sequentialization techniques usually assume sequential consistency. A very different approach is possible with *symbolic partial-order encodings* that can logically encode the partial-order concurrency semantics of a program, e.g. [YGLS03, YGLS04, GYS04, BAM07, WKGG09, TVD10, SW10, BWB+11, AKT13]. Since this makes symbolic partial-order encodings more suitable for weaker forms of concurrency semantics, in this thesis I study symbolic partial-order encodings rather than symbolic sequentialization techniques.

While symbolic partial-order encoding techniques have recently been shown to be promising for interesting shared memory programs, such as parts of the Apache HTTP server, PostgreSQL database and Linux kernel [AKT13], this thesis gives theoretical and experimental results that highlight a new set of challenges faced by such

techniques. Furthermore, it shows how different solutions to these problems emerge if partial-order encodings are restructured into three separate theory-specific parts, namely selection, value and clock constraints. This separation opens up a fresh and fruitful algorithmic perspective on partial-order encodings because it allows us to investigate the problem from different angles, as described next.

I first uncover inefficiencies in the state-of-the-art partial-order encoding that can be attributed to the combination of selection and clock constraints (Chapter 2). For this, I devise experiments with sufficiently large benchmarks that show that the total analysis run-time is mostly dominated by the construction of the partial-order encoding rather than the SAT solver. After establishing the fundamental concept of partial strings (Chapter 3), I then show how to asymptotically reduce the number of construction steps for partial-order encodings of a certain form of weak memory concurrency that permits reorderings (Chapter 4). This new partial-order encoding can drastically reduce the run-time of the analysis prior to calling the SAT/SMT solver. I then investigate how theory-specific forms of conflict-driven learning in SMT solvers interact with the solving of selection, value and clock constraints (Chapter 5). This investigation opens up new insights into the exponential worst-case time complexity of partial-order encodings during SAT/SMT solving, and highlights the significance of the selection constraints that so far have received less attention in the literature. Finally, guided by the previous results, I show how to significantly improve a backtracking algorithm that checks the linearizability of sets and maps (Chapter 6). Altogether, these results are significant as they propose a paradigm shift in how to harness the computational power of SAT/SMT solvers for partial-order encodings.

1.1 Contributions

This thesis makes the following contributions to the field of automated software verification and testing.

Chapter 2 experimentally evaluates the state-of-the-art partial-order encoding in CBMC [AKT13] using a new concurrency benchmark suite that I devised by extracting the OpenCores Ethernet MAC hardware model from the QEMU virtual machine. The extracted QEMU hardware model is shown to be realistic enough to reproduce and fix a known concurrency bug in an older version of the corresponding Linux device driver. What makes the OpenCores Ethernet MAC so interesting from a concurrency perspective is the fact that the state of the device is subject to incoming Ethernet frames. This gives rise to a significant degree of concurrency. The experimental

results quantify the surprisingly large overhead of constructing the propositional formula that captures this concurrency, and shows that the total analysis run-time is mostly dominated by CBMC rather than the SAT solver. The benchmarks that made these measurements possible were developed in discussion with Michael Tautschnig, Celina Val, Liaho Liang, Tom Melham, Jim Grundy and Daniel Kroening, with whom I published our FMCAD 2013 paper [HTV+13]. Note that unlike parts of our FMCAD 2013 paper [HTV+13], the experimental results reported in this chapter can be reproduced for all major CBMC releases since FMCAD 2013, leading up to the most recent version, CBMC 5.2, which was released on 29 August 2015.

Chapter 3 shows how Ésik's monotonic bijective morphisms [É02] between labelled partial-orders provide an algorithmically interesting model-theoretic perspective on a recent unifying theory of concurrency by Hoare et al. [HMSW11, HvS12b]. In particular, I provide an NP-complexity proof for the refinement problem of labelled partial-orders, a least fixed point reduction theorem for a certain pomset language refinement problem of theoretical interest, an iterative decision procedure based on this theorem, and experimental results that quantify the number of iterations in the decision procedure and the effect of the underlying SMT solver. The results of this work are relevant for pomset languages [Pra86, Gis88] closed under least fixed point, sequential and concurrent composition operators as well as the exchange law, which is closely connected to the frame rule in separation logic [OPVH15]. While the results in this chapter have no immediate practical application for now, they are the first small step towards automatically reasoning about more than one set of partially ordered events at the same time. This generalized reasoning power may be particularly relevant for future symbolic techniques that need to infer invariants with respect to the partial ordering of events, rather than merely the traditional mappings from shared program variables to the set of their possible values. Part of this work was published in FORTE 2015 [HK15] and was guided by discussions with Peter Schrammel, Marcelo Sousa and Björn Wachter.

Chapter 4 refines the partial-order model of concurrency from the previous chapter by interpreting a particular form of weak memory concurrency in terms of certain partially ordered sets. I show that this interpretation can be characterized by three fundamental weak memory axioms by Alglave et al. which underpin extensive experimental research into weak memory architectures [AMSS11]. An important consequence of this characterization is an asymptotically smaller partial-order encoding that has only a quadratic number of partial-order constraints compared to the state-of-the-art cubic-size encoding. I implement the new quadratic-size partial-order

encoding in CBMC and show that the new implementation drastically speeds up the construction of the propositional formula such that nearly all of the total run-time is now used for SAT solving. I show that this can reduce the total run-time for sufficiently large problems, such as the OpenCores Ethernet MAC benchmark, where at most 30–40% of the total run-time was previously used for SAT solving. However, I also carefully set up experiments to expose performance problems. Most of this work was published in FORTE 2015 [HK15] and SMT 2015 [HHK15], for which I formalized the partial-order encodings under scrutiny. This involved invaluable discussions with Jade Alglave, Michael Tautschnig and Daniel Kroening, who are the authors behind the theory and tool [AKT13] that inspired the work in this chapter.

Chapter 5 describes work done in collaboration with Liana Hadarean and Tim King. It addresses fundamental questions about the efficiency of SMT-based partial-order encodings. For this, the chapter proposes a new challenge problem for the SMT community, whose solution is directly relevant to finding concurrency-related bugs in software. This concurrency challenge problem prompts a new theoretical result for establishing lower bounds on the size of DPLL($\mathcal{T}$) proofs of unsatisfiability. This theoretical result is significant because it hides the implementation details of most state-of-the-art SMT solvers, providing a lower bound diagnostic tool for SMT-LIB encodings. In particular, this theory predicts a factorial lower bound on the size of DPLL($\mathcal{T}$) proofs of unsatisfiability for the proposed concurrency challenge problem. This theoretical lower bound is confirmed experimentally. Moreover, the experiments reveal that the number of SAT conflicts grows exponentially faster than theory conflicts. This rules out current SAT solver technology as an easy fix to the concurrency challenge problem. In addition, the theoretical and experimental results in this chapter provide strong evidence for the obstacles that the current DPLL($\mathcal{T}$) framework poses for efficiently exploiting the structure that is inherent in partial-order encodings. Despite these problems, I show that a linear arithmetic version of the same problem can usually be solved much more efficiently by adding theory lemmas in the form of unbounded intervals to the input formula. This work was published in part in SMT 2015 [HHK15]. My contribution involved devising the interval theory lemmas and writing the initial draft of the paper in which I set out the problem, formalized the SMT-based partial-order encodings, and provided the experimental setup and results that underpin the theory for the lower bound proofs.

Chapter 6 describes a new compositional approach for checking the linearizability of concurrent data types such as sets and maps. This problem is described through a well-known formalization that uses a special class of partial-orders, called interval

orders. I present P-compositionality, which generalizes the idea behind Herlihy and Wing's locality principle [HW90] to operations on the same concurrent data type. I implement P-compositionality in a novel linearizability checker that is an adaptation of Lowe's implementation [Low15]. I describe various design considerations in implementing the algorithm, and experimentally evaluate it on over nine lock-free and/or wait-free implementations of concurrent sets, including Intel's TBB library. These experiments show that the new linearizability checker is one order of magnitude faster and/or more space efficient than the state-of-the-art algorithm by Wing, Gong and Lowe [WG93, Low15] on which this chapter is based.

Chapter 2

Experimental framework

Collaborators *This chapter is based on joint work published in [HTV+13] with Michael Tautschnig, Celina Val, Liaho Liang, Tom Melham, Jim Grundy and Daniel Kroening.*

2.1 Introduction

To date, symbolic bounded model checking techniques have been successfully used to analyze sequential low-level software where the size of arrays, loops and recursion is usually bounded due to the limited computing resources available on devices on which low-level software tends to run. Given these resource limitations, it may not be entirely obvious what could give rise to concurrency-related bugs.

This chapter explains one source of such bugs when developing low-level software that closely interacts with hardware. This hardware/software interaction is the basis for a new concurrency benchmark suite we develop. We explain why this concurrency benchmark is relevant for the microelectronics industry, and how we make the benchmark realistic by extracting a hardware model from the QEMU virtual machine. The resulting benchmark is shown to be expressive enough to reproduce and fix a known bug in an older version of a Linux device driver. Using our concurrency benchmark, we experimentally evaluate the state-of-the-art partial-order encoding in CBMC [AKT13], a bounded model checker that has been consistently evolving over the last decade [KCY03]. Our experiments reveal how the analysis in CBMC is severely limited by the size of the partial-order encoding, even before the SAT solver is called. This limitation is one of the main motivations behind the study of partial-order encodings in this thesis. To experimentally evaluate our work in Chapter 4 that addresses this limitation, we will use the experimental framework described here.

2.1.1 Relevant and realistic benchmark code

The relevance of our benchmark suite derives from the influx of semiconductor designs that run firmware on built-in micro-controllers to provide functionality that would formerly have been implemented in hardware. This trend is driven by factors that include the following:

- Extracting the control of complex devices and implementing it in firmware can cut development schedules while adding flexibility and survivability.
- By making on-chip devices more capable, work can be shifted away from the CPU, where performance is increasingly hard-won. The richer control required for more capable devices further drives the trend.

The microelectronics industry has to therefore increasingly develop and validate low-level software that closely interacts with hardware when designing a new product. The design and validation of these products, with the predictability needed to schedule fabrication and hit market windows, has become an acute challenge. The crux of the problem is that a hardware/software interface often marks the boundary between different threads of concurrent execution. This problem is exacerbated by the fact that a hardware/software interface is built up from nonstandard primitives that may include interrupts, memory-mapped I/O, and special-purpose registers. The situation is analogous to software before procedure-calling conventions were standardized. Finally, a hardware/software interface almost always marks a boundary between different teams, working in different parts of a company and having different educational backgrounds and skills. This increases the scope for misunderstandings. Therefore, the challenges faced by those implementing the two sides of a hardware/software interface are high—but so is the need to get it right.

An increasingly common approach to confronting this challenge is to employ so-called *virtual prototypes* [Tei12]. These are essentially software models (written in C or SystemC) of the hardware, so that the firmware can be developed and tested before silicon is available. Virtual prototypes are usually not cycle accurate and therefore afford a higher level of abstraction than more traditional approaches to formal hardware verification. In our FMCAD paper [HTV+13], we previously identified the open-source QEMU [Bel05] code base, together with Linux device drivers [CRKH05], as a rich source of characteristic examples to drive research into bit-accurate virtual prototyping. The reason why these examples are important for experimental research purposes is as follows.

By extracting benchmark code from the QEMU virtual machine and the Linux kernel, we can strengthen the realism of our experiments, since they retain essential characteristics of production code. This includes a specific division into hardware models and low-level software, which moreover originate from separate developer communities. Finally, since we do not write the code for the benchmarks, we avoid experimental bias. We show how to apply this research method to the OpenCores Ethernet MAC [Moh11].[1,2]

What makes the OpenCores Ethernet MAC so interesting and challenging is the fact that the state of the device is subject to incoming Ethernet frames. This gives rise to a significant degree of concurrency, and we therefore use the QEMU hardware model of the OpenCores Ethernet MAC as an exemplar of the kind of concurrency that needs to be handled by low-level software. Crucially, we confirm through the concrete execution of the combined hardware/software code that we can reproduce and fix a known concurrency-related bug in an older version of the Linux device driver. This provides evidence that the hardware model in our benchmark suite is expressive enough for a meaningful analysis of the OpenCores Ethernet MAC hardware/software interface. Using this exemplar, we can therefore setup a relevant and realistic experimental evaluation of CBMC's partial-order encoding. We next describe this experimental setup from a high-level perspective.

2.1.2 Overview of experimental setup

In our experiments with the OpenCores Ethernet MAC, we concentrate on the functionality of receiving Ethernet frames. Each incoming frame is called an *RX frame*. The hardware can be configured such that RX frames trigger hardware interrupts. A unique characteristic of our virtual prototype is that it runs each interrupt handler as a concurrent thread. The main verification goal for us is to check that we have correctly modelled this asynchronous firing of interrupts on incoming RX frames. For this, we add run-time assertions to the code which are given formal meaning through a bit-precise execution semantics [PICW04] in CBMC. We collectively refer to these run-time assertions as *properties*. We mostly focus on MAC-specific concurrency-related properties that we check with respect to a small test harness that calls the QEMU hardware model.

The extracted QEMU hardware model is around 1,000 lines of code written in

[1] http://git.qemu.org/?p=qemu.git;a=blob;f=hw/net/opencores_eth.c

[2] https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/devicedrivers/net/ethernet/ethoc.c

C. We run a locally modified version of CBMC 5.2. The purpose of our experiments is to quantify the time and space requirements of the state-of-the-art partial-order encoding in CBMC. For this purpose, we measure the total run-time of CBMC and the SAT solving times for MiniSAT 2.2 and Glucose 4.0. The data points for both solvers are shown to increase the confidence in our experimental results, and are, in fact, crucial for the experiments in Section 4.4.3 later on.

Contributions The main contributions of this chapter are twofold:

1. We provide a sufficiently small, yet relevant and challenging, concurrency benchmark by extracting the hardware model of the OpenCores Ethernet MAC from the QEMU virtual machine. We show that the extracted QEMU hardware model is realistic enough to reproduce and fix a known bug in an older version of the OpenCores Ethernet MAC Linux device driver. This is evidence for the realism of our benchmark code.

2. Given this new concurrency benchmark, we experimentally evaluate the state-of-the-art partial-order encoding in CBMC. Our experimental results quantify the disproportionately large overhead of constructing the propositional formula before the SAT solver is called, and show that the overall analysis run-time is in fact dominated by CBMC rather than the SAT solver. With the necessary patches applied, we can reproduce our experimental results for all major CBMC releases over the last two years, leading up to the most recent version, CBMC 5.2, which was released on 29 August 2015. In each of these releases, we closely collaborated with the core developers of CBMC (Michael Tautschnig and Daniel Kroening) to ensure that our experimental results are not merely an artifact of an "engineering problem" in CBMC. This gives us the necessary scientific footing for studying and improving the state-of-the-art partial-order encoding later on in this dissertation.

Organization The rest of this chapter is organized as follows. We first explain the QEMU hardware model of the OpenCores Ethernet (Section 2.2). Following our research method, we then explain and reproduce a known bug in an older version of the corresponding Linux device driver using the QEMU hardware model (Section 2.3). Finally, we experimentally evaluate the state-of-the-art partial-order encoding using the OpenCores Ethernet MAC code and CBMC (Section 2.4).

2.2 OpenCores Ethernet MAC hardware model

In this section, we explain important aspects of the hardware model of the OpenCores Ethernet MAC that we have extracted from the QEMU virtual machine, and what modifications we have made to the hardware model.

From a hardware/software validation perspective, the purpose of the hardware model is to capture the hardware side of interactions between the Linux device driver and the MAC. This includes modelling complex, ad-hoc side-effects characteristic of low-level devices. The design of the QEMU hardware model of the OpenCores Ethernet MAC follows closely the hardware data sheet [Moh11], including the division into the following components:

- the *Media Access Control* (*MAC*) module which controls the transmission and receiving of Ethernet frames;
- the *Media Independent Interface* (*MII*) Management module which controls the device on the physical medium and gathers status information from it.

Informally, the MII enables the MAC to interface with the PHY, the physical layer transceiver which connects the MAC to a physical medium such as an optical fiber. Note that the details of the PHY are omitted from QEMU; instead, the hardware model features a procedure with which the arrival of an RX frame can be triggered.

At its core, the MII has six 16-bit registers and the MAC consists of 21 status and control registers, each 32 bits long. These registers are represented by two fixed-size arrays of `uint_16` and `uint_32` integers, respectively. The QEMU developers seemed to have opted for this array-based design because they wanted to encapsulate the reading and writing of these registers in C functions that are called through function pointers which are stored inside of arrays whose index corresponds to a particular register being accessed. This index is computed based on the hardware address that identifies a register. For example, the hardware data sheet [Moh11] specifies the hardware address of the MAC's interrupt source register to be `0x04`. In the hardware model, such a hardware address is divided by four to compute the index into the MAC's register array. Therefore, the index for the interrupt source register is 1. Based on this index, the hardware model retrieves a function pointer from an array. By calling the pointed to function, the interrupt source register will be accessed. This indirection achieves a certain separation of concerns but makes the QEMU code harder to analyze.

In addition to the registers, the MAC features 128 buffer descriptors, each of which contains information about a single frame, including its status and the address to an associated DMA buffer where the data is stored. A buffer descriptor that contains information about an RX frame is called an *RX buffer descriptor*. All buffer descriptors and the majority of MAC registers can be read and written by the Linux device driver as well as the hardware model, excluding a handful of MII status bits that can be only updated by the hardware model. This leads to a proliferation of possible device states that are encapsulated by the QEMU hardware model.

Conventionally, QEMU is a sequential program. We want to selectively introduce concurrency into the QEMU hardware model in order to capture the fact that hardware interrupts need not run in the order in which they were fired. We accomplish this as follows:

- Changes to the state of the hardware model can be invoked through thread-safe procedures. To achieve thread-safety, each procedure in the hardware model is defined to run atomically from all the others.

- When a hardware interrupt fires, the interrupt handler is spawned as a separate concurrent thread. This has the benefit that the order in which interrupt handlers run is unspecified.

The way we model the firing of interrupts must accommodate the fact that each procedure in the hardware model is made atomic by means of CBMC-specific source code annotations that let CBMC optimize the analysis. The problem is that this optimization prevents threads from being spawned inside an atomic procedure. The solution is to split the firing of an interrupt into two steps: set a flag inside the atomic region to indicate that an interrupt should be fired, then, check the flag outside the atomic region to spawn the thread if necessary.[3] To implement this, we statically analyzed the calling sites of the procedure that spawns the thread for the interrupt handler and we assume that the flag can be safely read without holding a lock.

2.2.1 Hardware model from QEMU

QEMU is designed for hardware virtualization, not experiments in formal validation, and extracting usable stand-alone hardware models from QEMU code is not entirely straightforward. To give an idea of what is involved, we briefly sketch some aspects

[3] https://github.com/ahorn/benchmarks/commit/3de4ec1f0d237d43fee43ff5b1cb237050

of the QEMU architecture, before going on to present the properties we check in the hardware model.

By default, QEMU accelerates dynamic code translation through just-in-time compilation [Bel05]. As a first step in extracting the hardware model, we bypassed this dynamic code translation through a feature called QTest [LWF+15], a client-server architecture made available to QEMU developers to facilitate testing of hardware models. QTest provides a helpful starting point for understanding the most essential dynamic and static dependencies of the code.

The code of the QEMU virtual machine is written in C, and can be broadly divided into boards, each of which consists of device and bus models. Communication between device models can occur only through bus models. This is the basis for a modular design, implemented through a QEMU-specific factory and service locator pattern known as QDev [Bon11].

QDev organizes hardware models into a dynamic tree data structure that relies on a QEMU-specific object model called QOM [Lig11]. In essence, QOM seeks to extend C with object-oriented programming features, including a form of class hierarchy. To achieve this, QOM stores information about its internal C structures in `glib` trees and hash tables. An instantiation of such a structure is called an object. Function pointers serve as methods to objects. QEMU's physical memory management architecture determines which object methods of a device model are called when memory regions are accessed by the guest operating system [KRB+15].

For our benchmarks, we extracted the stand-alone C hardware model by excluding all physical memory management code and dependencies on QDev and QOM. This was done through a somewhat laborious process of careful slicing and approximation of essential features.

A few additional simplifications were made to the QEMU code. Since the SAT solver has no array logic built in, we reduced the number of buffer descriptors in the hardware model to eight. Moreover, we suppressed interrupts on changes of the interrupt mask because this functionality appears to be QEMU-specific and not part of the Ethernet MAC itself.[4] Finally, we used global variables in setting up the benchmark code because it (perhaps rather paradoxically) simplifies the pointer analysis in CBMC.[5]

[4] https://github.com/ahorn/benchmarks/commit/5b705ef1de601d3513c558d319950ef8fc
[5] https://github.com/ahorn/benchmarks/commit/2ca2a7c822f7b1345da8abbd5eb937c92b

2.2.2 Hardware model properties

The hardware model is sufficiently complex to create interesting MAC-specific properties that can be checked independently from the Linux device driver. We have created a total of five such hardware model properties, all of which are expressed by checking the content of certain MAC registers. We expect all of the following properties to hold, except the third one for which CBMC is expected to find a counterexample:

(HW.1) We reset the MAC and check the reset value of essential MAC registers as defined in [Moh11, Section 3]. For the remaining properties, we ensure that the MAC and MII registers have been reset and that at least one RX buffer descriptor has been setup.

(HW.2) Initially, we check that the MAC can receive RX frames.

(HW.3) After enabling interrupts for RX frames and triggering the arrival of a single RX frame, we check that the interrupt handler has fired. This property can fail because the interrupt handler is spawned as a separate thread.

(HW.4) We repeat the previous check but this time we wait for all interrupts and interrupt handlers to finish, thereby ensuring that property (HW.3) holds.

(HW.5) Finally, we use a MAC configuration register to adjust the size of the available number of RX buffer descriptors to one. We then trigger two incoming RX frames. Since we test the hardware model in isolation from the Linux device driver, the only available RX buffer descriptor will be filled up by the first RX frame. We therefore check that the 'busy' status bit in the interrupt source register is set to 1, confirming that the second RX frame was disregarded due to a lack of RX buffer descriptors.

2.3 Ethernet MAC hardware/software interface

How do we know that hardware model is realistic? This section answers this question by reproducing a known bug in the Linux device driver through the concrete execution of the combined hardware/software benchmark code. For this, we explicitly time the computational steps in the code. This method objectively shows that our OpenCores Ethernet MAC hardware model benchmark is relevant and realistic.

To understand the bug, we take a closer look at two different modes of operation and how they affect the hardware/software interface. In the so-called 'interrupt

mode' of operation, the device generates a hardware interrupt for each RX frame. When interrupts are disabled but RX frames should still be processed, called 'polling mode', the Linux device driver polls for incoming data. A noteworthy complication in the hardware/software interface of the OpenCores Ethernet MAC is that the Linux device driver can switch between interrupt and polling mode to improve performance [SOK01]. Similar techniques are used for block devices with high data throughput, such as solid state drives. Switching between polling and interrupt mode is known to be error-prone, so the combined hardware/software code is a good exemplar for concurrency bugs due to interrupts in a producer-consumer scenario. We next explain one such bug and how the Linux developers fixed it.

2.3.1 Concurrency bug in Linux device driver

To reproduce the concurrency bug in the older version of the Linux device driver, we need to orchestrate the timing of low-level bit operations on MAC registers, which we first explain from the perspective of the Linux device driver and then from the hardware's point of view.

The hardware/software interface protocol for managing hardware buffers works as follows. The Linux device driver sets the 'empty' status bit in an RX buffer descriptor to 1 when the associated buffer can be overwritten by the hardware. Such an RX buffer descriptor is said to be *empty*. The hardware, in turn, clears the 'empty' bit to signal to the Linux device driver that the DMA buffer associated with the buffer descriptor contains a new RX frame. Despite its apparent simplicity, this communication protocol is error-prone when the Linux device driver switches from polling to interrupt mode due to concurrently incoming RX frames, as explained next.

Suppose there is at least one empty RX buffer descriptor. The Linux device driver switches from polling to interrupt mode as soon as it detects no new RX frames. To do this, it reads the 'empty' status bit of the next available RX buffer descriptor. Suppose the current buffer descriptor is empty. In this case, the version of the Linux device driver with the bug continues by clearing all RX interrupt sources before reenabling all RX interrupts. Unfortunately, this can result in RX frames being delayed or even dropped. Figure 2.1 shows an example, in which an RX frame arrives just after the check for new RX frames but before the RX interrupt sources are cleared. This RX frame will not trigger an interrupt until another one arrives. In fact, if there are no other ones, the delayed RX frame is not even promoted to the socket layer. This happens when the Linux device driver is stopped, say due to standby.

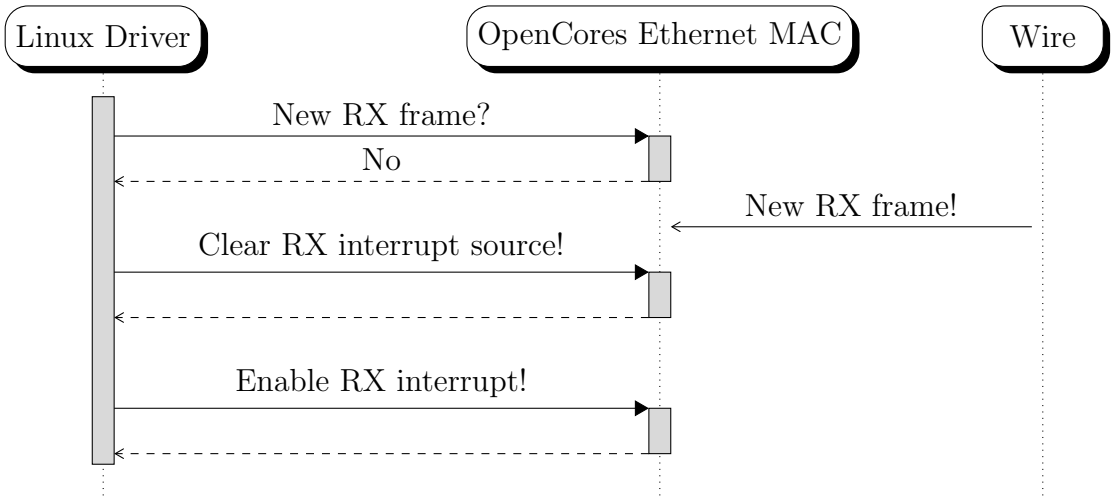


Figure 2.1: Incorrect handling of an empty RX buffer descriptor can cause data loss.

Figure 2.2 illustrates the way this problem manifests itself in the OpenCores Ethernet MAC device. As before, we assume that the Linux device driver is intially in polling mode and let us assume all buffer descriptors are empty. The former is depicted in Figure 2.2a by the 0_x bit in the interrupt source register and the latter is represented by `r` and `w` pointers which both point to the beginning of the array of buffer descriptors. Assume a new RX frame arrives, thereby changing the interrupt source from 0_a to 1_b and causing the first buffer descriptor to be non-empty (Figure 2.2b). Importantly, the Open Cores Ethernet MAC always sets the interrupt source register as new RX frames arrive (notice how 1_b has become 1_c in Figure 2.2c even though the `r` pointer remains unchanged). The Linux device driver reads one non-empty buffer descriptor, changing it to be empty again (Figure 2.2d). But simultaneously new RX frames can arrive (Figure 2.2e). As before, the Linux device driver continues to consume these until it detects that there are no other RX frames to consume (Figure 2.2f). So assume it initiates a procedure now to switch to interrupt mode. However, a fraction of a second later a new RX frame arrives. This changes the interrupt source from 1_d to 1_e and results in one new non-empty buffer descriptor (Figure 2.2g). But since the Linux device driver is not in interrupt mode yet, it fails to detect this intermittent RX frame and continues by clearing the interrupt source register (see 0_f in Figure 2.2h), before re-enabling interrupts (see 1_y in Figure 2.2i). Once the hardware has reached this state, an interrupt is only raised once another RX frame arrives, which is a problem because the RX frame will go unnoticed until that said interrupt is raised. Crucially, we reproduced this problem by running the

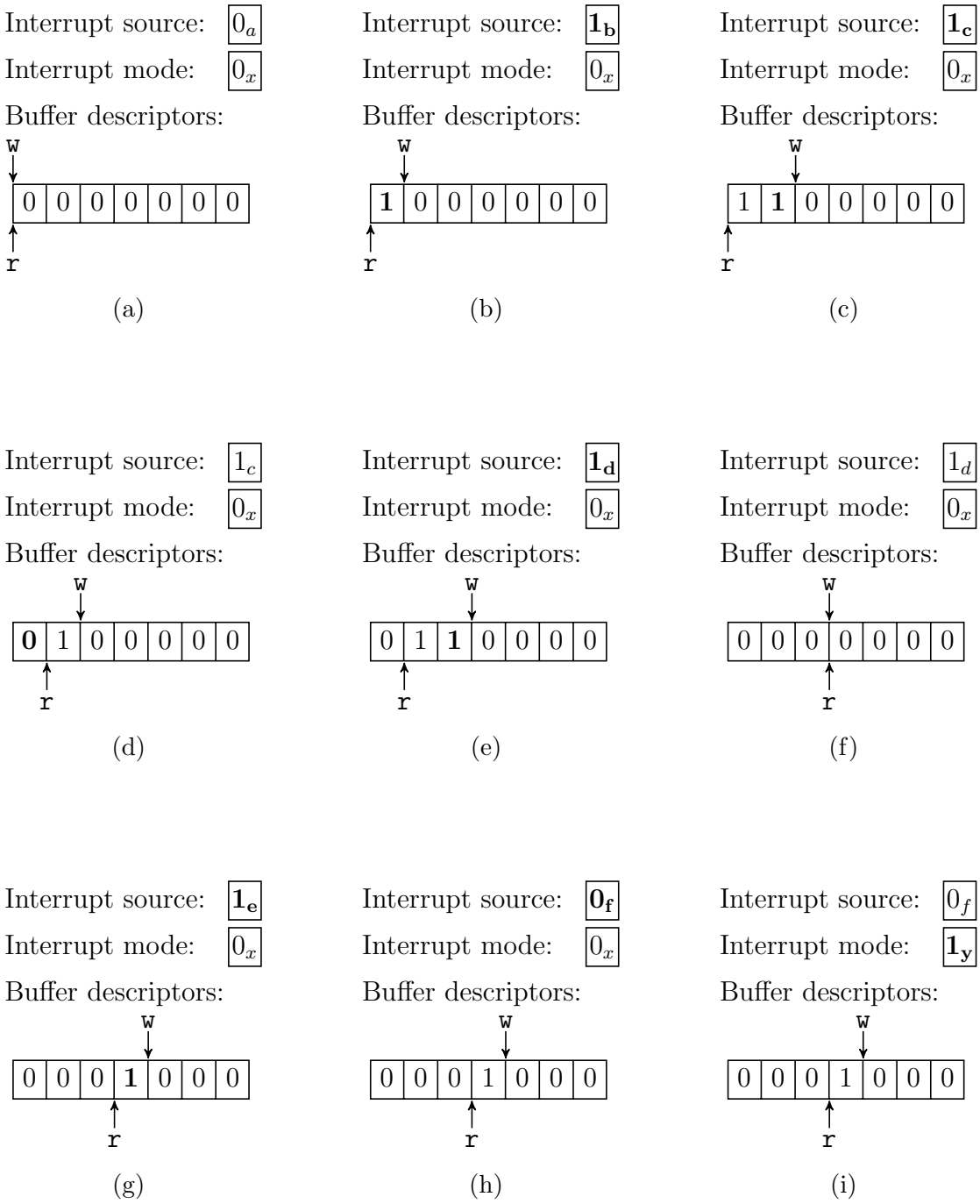


Figure 2.2: Concurrency bug in the old OpenCores Ethernet MAC Linux driver.

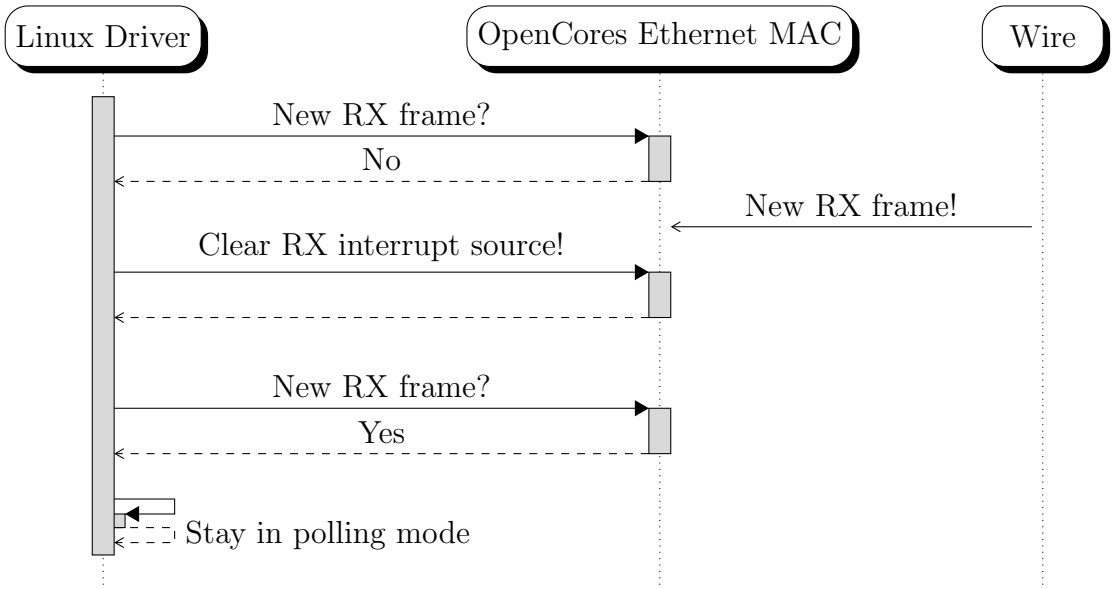


Figure 2.3: A second buffer descriptor check, after the RX interrupt sources have been cleared, detects intermittent RX frame arrivals.

Linux device driver using the extracted QEMU hardware model. Doing so, we found that the following property is sufficiently strong to expose this concurrency bug:

(SW.1) Provided the Linux device driver configures the MAC receiver such that there exists at least one empty RX buffer descriptor, the Linux device driver must eventually process every RX frame. At the very latest, when it is stopped, all RX frames must have been processed.

The crux of this property is that the Linux device driver must detect any potentially lost frames. Figure 2.3 shows how the Linux developers (Jonas Bonn and David S. Miller) fixed the bug in the Linux 2.6.38 kernel release.[6] The idea behind their fix is to check whether any new RX frames have arrived after the RX interrupt source has been cleared, thereby ensuring property (SW.1) is satisfied. We denote with 'SW*.1' the same property as (SW.1) except that we check it against modified benchmark code in which all loops and recursion procedures have been removed. This is only useful for experiments in which we expect the anlaysis to timeout or run out of memory.

Since our hardware model is executable, we confirmed that this known bug in the Linux device driver can be reproduced by concretely executing the combined hardware/software code. The concrete execution spawns interrupt handlers as regular

[6] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=20f70ddd6558a39a89dba4af675686c5a8dbd7b3

`pthreads` and we manually insert sleep instructions into the code to expose the bug. This sanity check serves as evidence that our MAC hardware model is expressive enough to find real bugs. However, for the symbolic analysis, similar to the hardware model, we reduce the maximum number of DMA buffers to eight and shrink their sizes to at most two bytes.

2.4 Experiments with the state-of-the-art partial-order encoding

For our experiments, we validate the properties of the QEMU hardware model and Linux device driver discussed in Section 2.2 and 2.3, respectively. We performed all experiments on a 64-bit machine running Linux 3.17.4 with eight Intel Xeon 3.07 GHz cores and 48 GB of physical main memory. We do not impose any timeout limit or non-physical memory quota. For the analysis, we run a modified version of CBMC 5.2, released on 29 August 2015.[7] The required modifications can be automatically performed by applying ten source code patches by Michael Tautschnig. These patches can be found alongside the source code repository for the OpenCores Ethernet MAC benchmark, which we have made publicly available online.[8] We use MiniSAT [EB05] (version 2.2) and Glucose [AS09] (version 4.0). We measure the memory footprint and the total run-time of CBMC using the 'runlim' tool (version 1.7) by Armin Biere and Toni Jussila.[9]

Our experimental results are given in Table 2.1. With one exception, each row in Table 2.1 corresponds to a single property being checked, except for the 'HW.1–5' benchmark where we check all of the hardware model properties collectively. We explain the metrics in each column in turn. The total run-time reports how many minutes it takes CBMC to solve a particular problem instance. This includes the SAT solving times, which are reported separately (also in minutes). The third column reports the memory usage in megabytes and shows 'MEMOUT' whenever either CBMC or the SAT solver exhausted all 48 GB of available memory.

The remaining five columns report metrics that are independent of the SAT solver but still give a sense for the difficulty of each benchmark. We report the maximum number of reads and writes performed by the program under scrutiny on one of the structs that either encapsulates the state of the hardware model or Linux device

[7] https://groups.google.com/d/msg/cprover/EN6LUjQMO74/anlHJVcJBAAJ
[8] https://github.com/ahorn/benchmarks/tree/master/opencores-ethernet-mac
[9] http://fmv.jku.at/runlim/

	Glucose 4.0							
	Time (*min*)			Max memory accesses			CNF Formula	
Property	Total	SAT	Memory (*MB*)	# reads	# writes	# SSA steps	# variables	# clauses
(HW.1)	0.8	0.6	804.5	8	228	212,182	723,406	3,880,031
(HW.2)	26.4	4.9	6201.8	79	233	2,273,594	6,675,618	48,268,314
(HW.3)	23.4	2.2	6089.9	74	234	2,162,355	6,788,467	46,743,360
(HW.4)	28.3	6.1	6340.0	78	234	2,273,905	6,885,264	48,844,791
(HW.5)	54.1	10.8	9051.8	108	236	3,268,334	9,317,200	70,228,109
(HW.1–5)	661.5	N/A	MEMOUT	356	250	15,179,328	N/A	N/A
(SW*.1)	445.0	N/A	MEMOUT	N/A	N/A	15,599,440	N/A	N/A
	MiniSAT 2.2							
(HW.1)	0.7	0.5	663.5	8	228	212,182	723,406	3,880,031
(HW.2)	32.1	10.9	7244.6	79	233	2,273,594	6,675,618	48,268,314
(HW.3)	33.2	12.3	7000.4	74	234	2,162,355	6,788,467	46,743,360
(HW.4)	36.7	14.8	7288.3	78	234	2,273,905	6,885,264	48,844,791
(HW.5)	74.7	32.0	10351.3	108	236	3,268,334	9,317,200	70,228,109
(HW.1–5)	651.7	N/A	MEMOUT	356	250	15,179,328	N/A	N/A
(SW*.1)	132.7	N/A	MEMOUT	N/A	N/A	N/A	N/A	N/A

Table 2.1: Experimental results for the OpenCores Ethernet MAC benchmark using the state-of-the-art partial-order encoding in CBMC.

driver. The 'SSA steps' column lists the number of equations in *single static assignment form* (*SSA*) that CBMC constructs before the SAT solver's search procedure is called. Informally, the number of SSA steps is proportional to the amount of memory that CBMC has to manipulate in order to construct an equation that represents the program under scrutiny. Once this equation has been constructed, CBMC converts it into a formula in *conjunctive normal form* (*CNF*). The total number of propositional variables and clauses in the fully generated CNF formula are reported in the last two columns, if applicable.

2.4.1 Description of experimental results

Our experimental results quantify the effect of the SAT solver and the time it takes CBMC to construct the equation for the property before the SAT solver is called. In 3 out of 6 cases, we get a 2–3X speedup in SAT solving times when using Glucose rather than MiniSAT, and in one case, property (HW.3), the speedup is almost by a factor of 5. This difference is significant enough to justify the importance of experiments that feature more than one SAT solver.

The faster Glucose solving times are reflected in shorter total run-times of CBMC, which rules out slowdowns in the way CBMC interacts with the SAT solvers (as expected since both SAT solvers have the same C++ interface). Most surprisingly and importantly, at least 60% of the total run-time is spent in CBMC rather than the

SAT solver. In fact, in the case of Glucose, this gap increases to 70%. This means that it takes more time for CBMC to construct the propositional formula than for the SAT solver to check its satisfiability, if applicable. This problem stems from the fact that the number of SSA steps in most of our benchmarks is several orders of magnitude larger than in all previously reported experiments [AKT13], even if we exclude the 'HW.1–5' and 'SW*.1' benchmarks that have 3–4 times more SSA steps than the remaining benchmarks.

In addition to run-time information, our experiments give clues to the memory consumption. In all cases except 'HW.1', Glucose consumes approximately 1 GB less memory than MiniSAT. Despite this, irrespective of the choice of SAT solver, the 'HW.1–5' and 'SW*.1' run out of memory after 10 hours. In the case of Glucose and the 'HW.1–5' benchmark, the OS signal that terminates CBMC indicates that the out-of-memory occurs inside the SAT solver. To more closely inspect this, we annotated the function that is called by CBMC when adding a clause to Glucose. We found that a disproportionately large fraction of the total run-time (more than 99%, so most of the 10 hours) is spent adding CNF clauses to the SAT solver.

2.4.2 Analysis of experimental results for subsequent work

To set the stage for later parts of this dissertation, we emphasize the following three main conclusions that can be drawn from our experiments with the state-of-the-art partial-order encoding. First, we find that the run-time depends significantly on the SAT solver. While this is perhaps not too surprising, it is currently not standard practice in the literature to run experiments with different SAT solvers. In this dissertation, prompted by the experimental results from the previous subsection, we shall deliberately run experiments on a broader spectrum of SAT solvers. Second, our experiments reveal the importance of measuring not only the total run-time of the analysis but also the SAT solving time. This has uncovered that the total run-time is dominated by CBMC rather than the SAT solver, including a case where CBMC runs out of memory. This may be attributed to the third and last fact: the OpenCores Ethernet MAC benchmark pushes partial-order encodings into uncharted territory because the number of SSA steps in this benchmark is significantly higher than in previous experiments [AKT13]. This warrants our later investigations into a partial-order encoding that can asymptotically reduce this number.

2.5 Concluding remarks

In the conventional approach to formal hardware verification, hardware models are essentially state-transition systems: a formal model is given that determines or constrains the next state of registers/memory in relation to the current state and inputs. In this chapter, we followed an approach that is particularly suitable for early phases of the design because it need not be cycle accurate and provides a higher-level, event-driven software abstraction of the hardware, focussed on concurrent interactions at the hardware/software interface.

We have used our modelling technique to derive a new concurrency benchmark that features code from the QEMU virtual machine and the Linux kernel. Our benchmark is particularly suitable for CBMC because, as its name suggests, it is primarily designed with C code in mind. Despite the fact that we used the most up-to-date version of CBMC, our experimental results provide strong evidence that CBMC spends more time constructing the partial-order encoding than it takes to solve it, or CBMC exhausts the available memory resources even before the SAT solver begins its search. This calls for a new partial-order encoding that can either speed up the formula construction or lower the memory footprint of CBMC. In Chapter 4, we give such an encoding.

2.6 Bibliographic notes

We give detailed related work for partial-order encodings in Section 4.6. Here, we only point to related work concerning the general space of validating hardware/software interfaces since this problem justifies the importance of our OpenCores Ethernet MAC benchmark.

Most research on verification of low-level software has focused on operating systems and device drivers, with some prominent successes [BLR11, CDE08, CE05, CKL04, GKS05, KEH+09]. There has also been some work on formal analysis of assembly code [CFF+06] and even binary device drivers have been analysed [KCC10]. There is, of course, a large body of literature on design and verification of embedded systems at a higher level [LMPS05].

The experiments in this chapter are a precursor to formal co-validation where the goal is to analyze the interactions between low-level software and on-chip hardware, e.g. [KLM+02]. There has been some related research in this area using bounded model checking [GKD06] and interval property checking [NWSK11]. The emphasis

of both these efforts is on machine instructions and cycle-accurate hardware models, while ours aims at early validation before a cycle-accurate model is available. This is reflected in the fact that the work of [NWSK11] targets Verilog code, while ours revolves around higher-level models in C. Earlier work also analyzed a more non-deterministic C model through abstract interpretation [Mon07], but with less sophisticated support for concurrency. More recently, an automata-theoretic co-verification technique has been applied to PCI device drivers [LXB+10].

Other related work has used symbolic simulation and SMT to check equivalence between a software reference model and a system containing (restricted) C code that invokes data computations on reconfigurable streaming hardware modelled in Java [TBL12]. Hardware/software concurrency is not represented; interaction is modelled by synchronous calls from the software into an API that loads and runs the streaming hardware designs. The aim is to establish correctness of the dataflow computations in hardware/software co-designs. By contrast, our modelling approach ultimately seeks to uncover bugs in hardware/software systems that interact through concurrent, imperative modification of shared state.

Given our modeling of interrupts in the hardware model, it would be interesting to experiment with rely/guarantee techniques for asynchronous programs, e.g. [GNK+15]. However, [GNK+15] is not directly applicable here since the code we have extracted from QEMU has no formally or informally stated pre-condition from which it would be possible to automatically derive rely/guarantee predicates.

Chapter 3

A partial-order model of concurrency suitable for SAT/SMT

3.1 Introduction

In this chapter, we start our investigation of partial-order encodings by looking through the lenses of a mathematical structure that has recently received much attention due to a new theory of concurrency by Hoare et al. [HMSW11, HvS12b], known as *Concurrent Kleene Algebra* (*CKA*). The significance of CKA is that it uses algebra to unify classical programming and process calculi, including those due to Hoare [Hoa69], Milner [Mil80], and Kahn [Kah87].

CKA is based on quantales, a special case of the fundamental algebraic structure of idempotent semirings. In addition, CKA combines the familiar laws of the sequential program operator (;) with a new operator for concurrent composition ($\parallel$). A distinguishing feature of CKA is its exchange law $(\mathcal{U} \parallel \mathcal{V}); (\mathcal{X} \parallel \mathcal{Y}) \subseteq (\mathcal{U}; \mathcal{X}) \parallel (\mathcal{V}; \mathcal{Y})$, which describes how sequential and concurrent composition operators can be interchanged. Intuitively, since the binary relation $\subseteq$ denotes program refinement, the exchange law expresses a divide-and-conquer mechanism by which concurrency may be sequentially implemented on a machine. While this is generally too restrictive for certain kinds of weak memory semantics, particularly when different threads can disagree on the order in which operations have taken effect, CKA is applicable when weak memory effects can be explained by relaxing the program order of a sequentially consistent system. Crucially, the exchange law, together with a uniform algebraic treatment of programs and their specifications, is key to unifying existing theories of concurrency [HMSW11, HvS12b], and is relevant for proving program correctness

using rely/guarantee proof rules [HMSW11]. Moreover, the exchange law is closely connected to the frame rule in separation logic [OPVH15]. This connection is remarkable because rely/guarantee reasoning is about interference, whereas separation logic centers on the idea of non-interference.

Despite its versatility, pure algebra alone cannot refute that a program is correct or that certain properties about every program always hold [HvS12a, HvS12b, HvSM+14]. This is problematic for theoretical reasons but also in practice because today's software complexity requires a diverse set of program analysis tools that range from proof assistants to automated testing. The solution is to accompany CKA with a mathematical model which satisfies its laws so that we can *prove* as well as *disprove* properties about programs.

One such well-known model-theoretical foundation for CKA is Pratt's [Pra86] and Gischer's [Gis88] partial-order model of computation that is constructed from so-called *labelled partially ordered multisets* (*pomsets*). Pomsets strictly generalize the concept of a string in finite automata theory by relaxing the total ordering of the occurrence of letters within a string to a partial-order. For this reason, pomsets are also known as partial words [Gra81]. Each occurrence of a letter is called an *event*, and the definition of pomsets in terms of partially ordered sets of events gives a natural way of defining sequential and concurrent composition through a generalized form of string concatenation and disjoint union, respectively. For example, $a \parallel a$ denotes a pomset that consists of two unordered events that are both labelled with the letter a. If we think of such letters as some action, then $a \parallel a$ could be interpreted as the concurrent execution of two a actions. By partially ordering events, pomsets form an integral part of the extensive theoretical literature on so-called 'true concurrency', e.g. [Pet66, Lam78, Gra81, NPW81, Pra86, Gis88], in which pomsets strictly generalize Mazurkiewicz traces [Maz87, BK92], and prime event structures [NPW81] are pomsets enriched with a conflict relation subject to certain conditions.

From an algorithmic point of view, the computational complexity of the *pomset language membership* (*PLM*) problem is NP-complete, and the pomset language containment problem belongs to an even higher complexity class [FKL93]. Importantly, these complexity results only apply to star-free pomset languages (without fixed point operators). In fact, the decidability of the equational theory of the pomset language closed under least fixed point, sequential and concurrent composition operators (but without the exchange law) has been only most recently established [LS14]; its complexity remains an open problem [LS14]. Yet another open problem is the decidability of this equational theory together with the exchange law [LS14].

These gaps are motivation to reinvestigate pomsets from an algorithmic perspective. To make this investigation precise, we adopt pomsets in the form of *partial strings* (Definition 3.2.1). Partial strings are different from pomsets [Pra86, Gis88] and partial words [Gra81] in one important way: partial strings are grounded on Ésik's *monotonic bijective morphisms* [É02], whereas pomsets and partial words are traditionally defined in terms of isomorphism classes [Gra81, Pra86, Gis88]. This difference matters here for three reasons: (i) isomorphisms are about sameness whereas the exchange law on partial strings is an inequation; (ii) since the partial string model of CKA features least fixed point operators, it is advantageous from an algorithmic point of view to define partial string operators irrespective of a representative in an isomorphism class; (iii) finally, and more importantly, the concept of monotonic bijective morphisms will be shown to appeal to decision procedures for binary relations.

We therefore opt for monotonic bijective morphisms as we construct a partial string model of CKA. As part of this construction, we lift many results of partial strings to a Hoare powerdomain in which programs (Definition 3.3.1) are defined by sets of finite partial strings, each of which is downward closed with respect to Ésik's monotonic bijective morphism. We choose this downward closure construction because it captures all possible happens-before relations of a concurrent system, i.e. all ways in which events can be ordered. In addition, this construction forms a complete lattice where the join operator can be interpreted as a form of mutual non-deterministic choice. While this Hoare powerdomain takes into account the difference between concurrency and non-deterministic choice, and has properties consistent with CKA, it is far from perfect. In particular, the downward closure over-approximates the behaviour of a concurrent system because it may include too many partial strings. In the next chapter, this over-approximation will be made more precise.

Throughout this chapter, our emphasis is on the ordering of events. This is justified by empirical data that shows that more than 30% of concurrency-related bugs are due to bad orderings of machine instructions rather than variable valuations [LPSZ08]. Some of these problems could be formalized through partial strings (or downward-closed sets thereof), and we show how they can be automatically solved by existing and highly optimized decision procedures such as SAT/SMT solvers. Towards this goal, we leverage decision procedures for quantifier-free equality logic and uninterpreted functions, one of the most fundamental first-order logics in SMT-LIB [BFT15]. This is the first small step towards automatically reasoning about more than one set of partially ordered events at the same time. We believe this generalized reasoning power will be relevant for future developments of symbolic techniques that need to

infer invariants with respect to the partial ordering of events, rather than merely the traditional mappings from shared program variables to the set of their possible values.

Contributions The main contributions in this chapter are threefold:

1. We prove that the partial string refinement problem is NP-complete, connecting our constructive interpretation of Ésik monotonic bijective morphism [É02] with the previously studied PLM problem [FKL93].

2. We give the theoretical basis for a decision procedure that can handle a fragment of CKA (Theorem 3.3.15). This is accomplished by exploiting a form of periodicity, thereby giving a mechanism for reducing a countably infinite number of events to a finite number. This result particularly caters to partial-order encoding techniques that can currently only encode a finite number of events due to the deliberate restriction to quantifier-free first-order logic, e.g. [AKT13].

3. We give a simple reference implementation of the decision procedure, and experimentally evaluate it. We give an optimization that is shown to reduce the memory footprint by up to two orders of magnitude.

Organization The rest of this chapter is organized into three sections. First, we recall familiar concepts on the theory of partial strings (Sections 3.2) on which the rest of this chapter is based. We then prove a least fixed point reduction result (Section 3.3). Finally, we evaluate our reference implementation (Section 3.4).

3.2 Partial-string theory

In this section, we adapt an axiomatic model of computation that uses partial-orders to describe the semantics of concurrent systems. For this, we recall familiar concepts (Definition 3.2.1, 3.2.4, 3.2.12 and 3.3.1) that underpin our mathematical model of CKA (Theorem 3.3.6). This model is the basis for subsequent results in Section 3.3.

3.2.1 Partial strings and their operators

The next concepts are well established yet fundamental to everything that follows.

Definition 3.2.1 (Partial string)**.** Denote by E a nonempty set of **events**. Let Γ be an **alphabet**. A **partial string** p is a triple $\langle E_p, \alpha_p, \preceq_p \rangle$ where E_p is a subset of

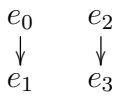


Figure 3.1: A partial string $p = \langle E_p, \alpha_p, \preceq_p \rangle$ with events $E_p = \{e_0, e_1, e_2, e_3\}$.

E, $\alpha_p \colon E_p \to \Gamma$ is a function that maps each event in E_p to an alphabet symbol in Γ, and $\preceq_p$ is a partial-order on E_p. When no ambiguity arises, we omit the subscripts.

For all events e, e' in E_p, we say e **happens-before** e' in p whenever $e \preceq_p e'$. Both events are said to **happen concurrently** in p whenever $e \not\preceq_p e'$ and $e' \not\preceq_p e$. Two partial strings p and q are said to be **disjoint** whenever $E_p \cap E_q = \emptyset$. A partial string p is called **empty** whenever $E_p = \emptyset$. Denote with P_f the set of all **finite partial strings** p whose event set E_p is finite.

Each event in the universe E should be thought of as an occurrence of a computational step, whereas letters in the alphabet Γ may be used to describe the kind of memory access performed by an event. For the most part of this chapter, we treat these letters as uninterpreted symbols (this will change in the next chapter where they distinguish reads and writes to particular memory locations). Typically, we denote a partial string by p, or letters from x through z. In essence, a partial string p is a partially ordered set $\langle E_p, \preceq_p \rangle$ equipped with a labelling function α_p. A partial string is therefore the same as a *labelled partial-order* (*lpo*). We draw finite partial strings in P_f as inverted Hasse diagrams (e.g. Figure 3.1) where we explicitly draw arrows to make clear the fact that they should be read from top to bottom. This is convenient since it can more easily convey the ordering of the instructions in a concurrent system. More generally, it is helpful to interpret the ordering between events as a form of happens-before relation [Lam78], a fundamental notion in distributed systems and formal verification of concurrent systems, e.g. [BOS+11, AMSS12]. We therefore often speak of the *happens-before relation* of a partial string p when we actually mean $\preceq_p$. In fact, when the partial string p is clear from the context, we just refer to the happens-before relation. We remark the obvious fact that the empty partial string is unique under component-wise equality.

Remark 3.2.2. *Since the concept of a partial string is closely related to that of a partial-order, it is interesting that Roscoe notes that the set of strict partial-orders over a fixed universe forms a dcpo [RHB97, p. 474].*[1] *He also mentions that it is far*

[1]Recall that a nonempty subset D of a partially ordered set is called **directed** if each finite subset F of D has an upper bound in D. For example, the set of all finite subsets of natural numbers,

from trivial to show that the maximal elements in this dcpo are the total orders on the universe, a fact that requires the Axiom of Choice.

Example 3.2.3. The partial string p in Figure 3.1 consists of four events in E_p. Since we treat the letters in the alphabet as uninterpreted symbols here, we leave the labelling function α_p unspecified for now. Note that e_0 happens-before e_1 in p, whereas both e_0 and e_2 happen concurrently in p because neither $e_0 \preceq_p e_2$ nor $e_2 \preceq_p e_0$.

We aim to abstractly describe essential aspects of concurrent systems by adopting the sequential and concurrent operators on labelled partial-orders [Gra81, Pra86, Gis88, É02] because they naturally describe the two operations that are inherent in these systems. By formalizing these operators, we can compose partial strings to form new ones. It is perhaps tempting to define these kind of compositions as a point-wise join operator. However, this would be problematic because the union of overlapping partial-orders is generally only a preorder that may violate antisymmetry. Fortunately, this can be remedied with a more sophisticated construction that ensures that the composed partial strings are disjoint. In particular, we resort to a form of coproducts to guarantee disjointness by construction.

Definition 3.2.4. Let $\mathbb{B} \triangleq \{0, 1\}$ be integers zero and one. For every set S and T, the **coproduct** of S and T, written $S + T$, is the set union of $S \times \{0\}$ and $T \times \{1\}$. Given two partial strings x and y, define their **concurrent** and **sequential composition** by $x \parallel y \triangleq \langle E_{x\parallel y}, \alpha_{x\parallel y}, \preceq_{x\parallel y}\rangle$ and $x; y \triangleq \langle E_{x;y}, \alpha_{x;y}, \preceq_{x;y}\rangle$, respectively, where $E_{x\parallel y} = E_{x;y} \triangleq E_x + E_y$ are coproducts such that, for all events $e, e' \in E_x \cup E_y$ and $i, j \in \mathbb{B}$, the following holds:

- $\langle e,\ i\rangle \preceq_{x\parallel y} \langle e',\ j\rangle$ exactly if $(i = j = 0$ and $e \preceq_x e')$ or $(i = j = 1$ and $e \preceq_y e')$,
- $\langle e,\ i\rangle \preceq_{x;y} \langle e',\ j\rangle$ exactly if $i < j$ or $\langle e,\ i\rangle \preceq_{x\parallel y} \langle e',\ j\rangle$,
- $\alpha_{x\parallel y}(\langle e,\ i\rangle) = \alpha_{x;y}(\langle e,\ i\rangle) \triangleq \begin{cases} \alpha_x(e) & \text{if } i = 0 \\ \alpha_y(e) & \text{if } i = 1. \end{cases}$

Example 3.2.5. Figure 3.2 illustrates the sequential and concurrent composition of two simple partial strings x and y as coproducts.

ordered by subset inclusion, is directed. A **directed-complete partial-order** (often abbreviated **dcpo**) is a partial-order with a bottom element and in which every directed set has a least upper bound. An example of a dcpo is the set of all partial functions ordered by subset inclusion of their respective graphs [DP02, pp. 180f].

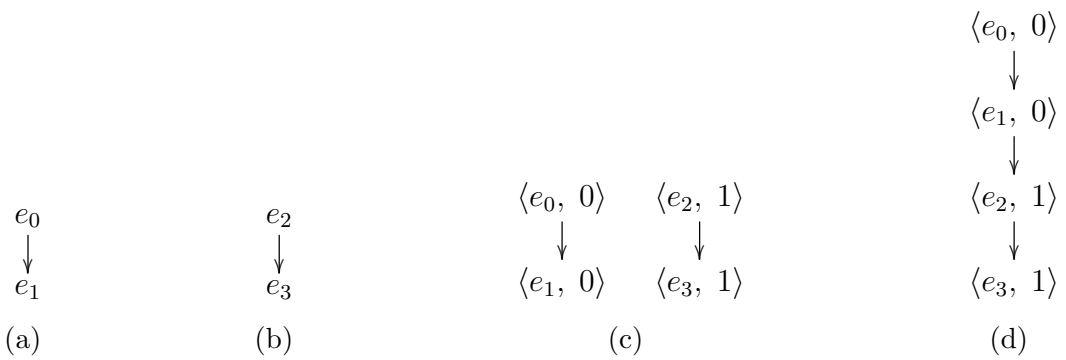


Figure 3.2: Let x and y be finite partial strings as shown in (a) and (b), respectively. Then (c) and (d) illustrate the coproduct for $x \parallel y$ and $x; y$, respectively.

For the concept of coproducts to be applicable in general, we require that the set of events E is countably infinite. Given sets S and T, their coproduct $S + T$ is like a 'constructive disjoint union'. This significantly shapes the kind of proofs we get about sequential (;) and concurrent ($\parallel$) composition because we can prove properties by calculation, i.e. computation. While mathematically not very interesting, we will shortly see how this matters computationally for a theory that aims at propositional logic or decidable fragments of first-order logic.

Definition 3.2.6 (Partial string isomorphism)**.** Let x and y be partial strings such that $x = \langle E_x, \alpha_x \preceq_x\rangle$ and $y = \langle E_y, \alpha_y, \preceq_y\rangle$. Then x and y are **isomorphic**, denoted by $x \cong y$, if there exists an order-isomorphism between x and y that preserves their labeling; equivalently, there exists a one-to-one and onto function (**bijection**) $f\colon E_x \to E_y$ such that, for all events $e, e' \in E_x$, the following holds:

1. $e \preceq_x e'$ if and only if $f(e) \preceq_y f(e')$, and
2. $\alpha_x(e) = \alpha_y(f(e))$.

Example 3.2.7. If we ignore labels for now and let p_i for all $0 \leq i \leq 3$ be four partial strings that each consist of a single event e_i, then $(p_0; p_1) \parallel (p_2; p_3)$ corresponds to a partial string that is isomorphic to the one shown in Figure 3.1 as well as Figure 3.2c.

More generally, it is easy to establish that we always get a new partial string when we use $\parallel$ or ;. That is to say, the set of partial strings is closed under both of our coproduct binary operators, and we can therefore speak of $\parallel$ and ; as *partial string operators*.

Proposition 3.2.8 (Partial string operators)**.** *For every partial string x and y, $x \parallel y$ and $x; y$ are also partial strings.*

Proof. Show that $\preceq_{x\|y}$ and $\preceq_{x;y}$ are partial-orders. By assumption, $\preceq_x$ and $\preceq_y$ are partial-orders. By case analysis, $\preceq_{x\|y}$ is reflexive, transitive and antisymmetric. Since $\preceq_{x\|y}$ is a subset of $\preceq_{x;y}$, it follows that $\preceq_{x;y}$ is reflexive and transitive. Let e, e' be events in $E_x \cup E_y$ and $i, i' \in \mathbb{B}$. Assume $\langle e,\ i\rangle \preceq_{x;y} \langle e',\ i'\rangle$ and $\langle e',\ i'\rangle \preceq_{x;y} \langle e,\ i\rangle$. Definition 3.2.4 implies $i = i'$. From antisymmetry of $\preceq_x$ and $\preceq_y$ follows $e = e'$. By point-wise equality, $\langle e,\ i\rangle = \langle e',\ i'\rangle$. We conclude that $x \parallel y$ and $x; y$ are partial strings according to Definition 3.2.1. □

The way coproducts shape proofs about properties of $\parallel$ and ; is best illustrated through an example. For this purpose, it follows a simple proof of the fact that $x \parallel y$ and $y \parallel x$ are isomorphic for every partial string x and y (whether finite or not):

Proposition 3.2.9 ($\parallel$-commutativity)**.** *For all partial strings x and y, $x \parallel y \cong y \parallel x$.*

Proof. To show that concurrent composition of partial strings is commutative, let $f\colon E_{x\|y} \to E_{y\|x}$ be a function from the events in $x \parallel y$ to the events in $y \parallel x$ such that, for all $e \in E_x \cup E_y$ and $i \in \mathbb{B}$, $f(\langle e,\ i\rangle) = \langle e,\ 1 - i\rangle$. Clearly f is bijective. It remains to show that f is a label-preserving order-isomorphism. Let $e' \in E_x \cup E_y$ and $i' \in \mathbb{B}$. Then

$$\begin{aligned} \langle e,\ i\rangle \preceq_{x\|y} \langle e',\ i'\rangle &\Leftrightarrow \langle e,\ 1-i\rangle \preceq_{y\|x} \langle e',\ 1-i'\rangle && \{\text{Definition 3.2.4 of } \parallel \text{ with } i = i'\} \\ &\Leftrightarrow f(\langle e,\ i\rangle) \preceq_{y\|x} f(\langle e',\ i'\rangle). && \{\text{Definition of } f\} \end{aligned}$$

Moreover $\alpha_{x\|y}(\langle e,\ i\rangle) = \alpha_{y\|x}(\langle e,\ 1-i\rangle) = \alpha_{y\|x}(f(\langle e,\ i\rangle))$. Hence $x \parallel y \cong y \parallel x$. □

We have given the details of the proof to draw attention to the fact that the label-preserving order-isomorphism acts as a witness for the truth of the statement. We call this a *constructive proof*. Along similar lines, it is not difficult to constructively prove that $\bot$ is the identity element (up to isomorphism) for both the sequential and concurrent partial string composition operators (Proposition 3.2.11). In fact, since it is a recurring theme for an algebraic property to hold for both operators, it is convenient to define the following:

Definition 3.2.10 (Bow tie)**.** For all partial strings x and y, denote with $x \bowtie y$ either concurrent or sequential composition of x and y. That is, a statement about $\bowtie$ is the same as two statements where $\bowtie$ is replaced by '$\parallel$' and ';', respectively.

For example, the next statement is equivalent to the following two isomorphisms: $x \parallel \bot \cong \bot \parallel x \cong x$ and $x; \bot \cong \bot; x \cong x$ for all partial strings x.

Proposition 3.2.11 ($\bowtie$-identity)**.** *For all partial strings x, $x \bowtie \bot \cong \bot \bowtie x \cong x$.*

Proof. Let $f\colon E_x \to E_{x\bowtie\bot}$ be a function such that, for all $e \in E_x$, $f(e) = \langle e, 0\rangle$. By Definition 3.2.4, $E_{x\bowtie\bot} = E_x \times \{0\}$ because $E_\bot \times \{1\} = \emptyset$. Clearly f is a label-preserving order-isomorphism, whence $x \bowtie \bot \cong x$. Similarly, $\bot \bowtie x \cong x$. □

3.2.2 Partial string refinement

As explained in Chapter 1, one defining characteristic of weak memory is that loads and stores may be reordered. This can cause programs to violate interleaving semantics and often requires low-level control over the ordering of instructions. While weaker forms of memory semantics can significantly increase execution efficiency, they can also lead to subtle and notoriously difficult to find concurrency-related bugs. It is therefore natural to ask how we can automatically compare two concurrent programs that are identical except perhaps from the ordering of their events. It is not difficult to see that this question can be particularly relevant for analyzing code that targets different computer architectures or code compiled with different compilers.

With this in mind, we formalize the refinement of partial strings such that we may leverage SAT/SMT solvers. For this reason, our formalism relies on Ésik's notion of monotonic bijective morphism [É02]:

Definition 3.2.12 (Partial string refinement)**.** Let x and y be partial strings such that $x = \langle E_x, \alpha_x \preceq_x\rangle$ and $y = \langle E_y, \alpha_y, \preceq_y\rangle$. A **monotonic bijective morphism** from x to y, written $f\colon x \to y$, is a bijective function f from E_x to E_y such that, for all events $e, e' \in E_x$, the following holds:

1. $\alpha_x(e) = \alpha_y(f(e))$, and
2. if $e \preceq_x e'$, then $f(e) \preceq_y f(e')$.

Then x **refines** y, written $x \sqsubseteq y$, if there exists a monotonic bijective morphism $f\colon y \to x$ from y to x.

The first condition says that the bijection is label-preserving and the second condition adds the requirement that it is monotonic. In effect, this ensures that we can disregard the identity of events but retain the notion of 'subsumption', cf. [Gis88]. The intuition is that $\sqsubseteq$ orders partial strings according to their determinism. In other words, $x \sqsubseteq y$ for partial strings x and y implies that all events ordered in y have the same order in x. This explains why we think of $\sqsubseteq$ as a refinement ordering between partial strings, which is similar to the dual of the "weakening order" in [BEEH15].

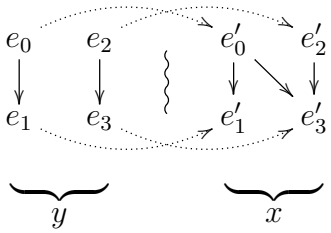


Figure 3.3: Two partial strings x and y such that $x \sqsubseteq y$ provided all the labels are preserved, e.g. $\alpha_x(e'_0) = \alpha_y(e_0)$.

Example 3.2.13. Figure 3.3 shows a monotonic bijective morphism from a partial string as given in Figure 3.1 to an N-shaped partial string that is almost identical to the one in Figure 3.1 except that it has an additional partial-order constraint, giving its N shape. One well-known fact about N-shaped partial strings is that they cannot be constructed as $x; y$ or $x \parallel y$ under any labelling [Pra86]. However, this is not a problem for our study, as will become clear after Definition 3.3.1.

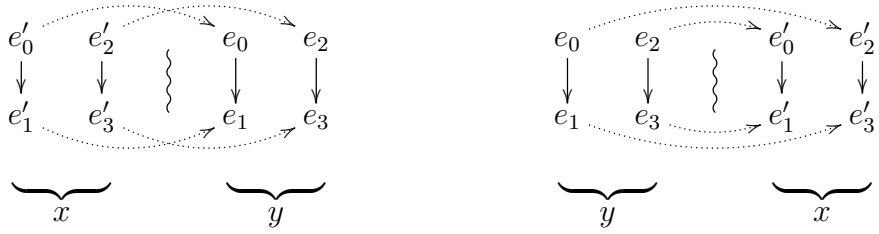
Remark 3.2.14. *For arbitrary partial strings, the refinement ordering $\sqsubseteq$ from Definition 3.2.12 is different from the containment of two pomset languages, cf. [FKL93]. For example, the languages of the partial strings x and y as shown in Figure 3.3 are contained in each other but $y \not\sqsubseteq x$, see also Pratt [Pra86, p. 13]. More accurately, Definition 3.2.12 implies pomset language containment, but not vice versa.*

The following important fact about $\sqsubseteq$ is clear since every finite number of function compositions preserves monotonicity and bijectivity.

Proposition 3.2.15. *$\sqsubseteq$ forms a preorder on the family of all partial strings.*

Proof. Let x, y and z be partial strings. Since the identity function is bijective, monotonic and label-preserving, $x \sqsubseteq x$, proving reflexivity. Assume $x \sqsubseteq y$ and $y \sqsubseteq z$. By Definition 3.2.6, there exist two monotonic bijective morphisms $f\colon y \to x$ and $g\colon z \to y$. By function composition, $f \circ g\colon z \to x$ is a monotonic bijective morphism, proving transitivity. □

Even though $\sqsubseteq$ is a preorder, it is generally *not* a partial-order unless we impose further restrictions on partial strings [É02]. In particular, the following proposition allows us to treat all *finite* partial strings as a partially ordered set because the refinement order $\sqsubseteq$ on P_f is antisymmetric under the partial string isomorphism from Definition 3.2.6, cf. Proposition 3.5 in [É02].



(a) (b)

Figure 3.4: The left and right show a monotonic bijective morphism $f\colon x \to y$ and $g\colon y \to x$, respectively. Thus $y \sqsubseteq x$ and $x \sqsubseteq y$. Let $h \triangleq g \circ f$. Then $h^2 = h \circ h$ is the identity function on E_x. Both f and g are isomorphisms, whence $x \cong y$.

Proposition 3.2.16 (Antisymmetry)**.** *For all finite partial strings $x, y \in \mathsf{P}_f$, if $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x \cong y$.*

Proof. Let $x, y \in \mathsf{P}_f$ be finite partial strings. Let $f\colon x \to y$ and $g\colon y \to x$ be monotonic bijective morphisms as witnesses for $y \sqsubseteq x$ and $x \sqsubseteq y$, respectively. Let $e, e' \in E_x$ be events. By Definition 3.2.6, it suffices to show $f(e) \preceq_y f(e')$ implies $e \preceq_x e'$. Assume $f(e) \preceq_y f(e')$. Let $h \triangleq g \circ f$. Define $h^1 \triangleq h$ and $h^{n+1} \triangleq h^n \circ h$ for all natural numbers $n \in \mathbb{N}$. Since E_x and E_y are finite, there exists k such that h^k is the identity function on E_x. Fix k to be the smallest such natural number. Assume $k = 1$. Then $g(f(e)) \preceq_x g(f(e'))$ by assumption and g's monotonicity. Since h is the identity function, it follows $e \preceq_x e'$, as required. Now assume $k > 1$. Since every finite number of function compositions preserve monotonicity, $h^{k-1}(g(f(e))) \preceq_x h^{k-1}(g(f(e)))$. Since h^k is the identity function, $e \preceq_x e'$, proving that f is an isomorphism. We conclude that $x \cong y$. □

Figure 3.4 illustrates the essence of the proof behind Proposition 3.2.16. And clearly its converse also holds (even for infinite partial strings) because a bijective monotonic morphism generalizes the concept of an isomorphism.

Proposition 3.2.17. *For all partial strings x and y, if $x \cong y$, then $x \sqsubseteq y$ and $y \sqsubseteq x$.*

Proof. Let x and y be partial strings. Assume $x \cong y$. Fix f to be an isomorphism from x to y. By Definition 3.2.6, f is a bijective monotonic morphism, and so is its inverse $f^{-1}\colon y \to x$, proving $y \sqsubseteq x$ and $x \sqsubseteq y$, respectively. □

As shown in Section 3.4, our notion of partial string refinement is particularly appealing for symbolic techniques of concurrency because the monotonic bijective morphism can be directly encoded as a first-order logic formula modulo the theory of uninterpreted functions. These formulas could be also translated into propositional

logic. Either way, such a symbolic partial-order encoding is fully justified from a computational complexity perspective, as shown next.

Proposition 3.2.18. *Let x and y be finite partial strings in P_f. The **partial string refinement** (PSR) problem — i.e. whether $x \sqsubseteq y$ — is NP-complete.*

Proof. Clearly PSR is in NP. The NP-hardness proof proceeds by reduction from the pomset language membership (PLM) problem defined in [FKL93]. Let Γ^* be the set of strings, i.e. the set of finite partial strings s such that $\preceq_s$ is a total order (for all $e, e' \in E_s$, $e \preceq_s e'$ or $e' \preceq_s e$). Given a finite partial string p, let $\mathfrak{L}_p$ be the set of all strings that refine p; equivalently, $\mathfrak{L}_p \triangleq \{s \in \Gamma^* \mid s \sqsubseteq p\}$. So $\mathfrak{L}_p$ denotes the same as $L(p)$ in [FKL93, Definition 2.2].

Let s be a string in Γ^* and P be a pomset over the alphabet Γ. Fix p to be a partial string in P. Thus s refines p if and only if s is a member of $\mathfrak{L}_p$. Since this membership problem is NP-hard [FKL93, Theorem 4.1], it follows that the PSR problem is NP-hard. So the PSR problem is NP-complete. □

As shown next, the two binary operators on partial strings are related in the expected way as already witnessed in the example of Figure 3.2.

Proposition 3.2.19 (Basic refinement)**.** *For all $x, y \in \mathsf{P}$, $x; y \sqsubseteq x \parallel y$.*

Proof. Let $e, e' \in E_x \cup E_y$, $i, i' \in \mathbb{B}$, and $f \colon E_{x\parallel y} \to E_{x;y}$ is a function such that $f(\langle e,\, i\rangle) = \langle e,\, i\rangle$. Clearly f is a bijection. By Definition 3.2.4, f preserves labels because $\alpha_{x\parallel y}(\langle e,\, i\rangle) = \alpha_{x;y}(\langle e,\, i\rangle) = \alpha_{x;y}(f(\langle e,\, i\rangle))$, and $\langle e,\, i\rangle \preceq_{x\parallel y} \langle e',\, i'\rangle$ implies $f(\langle e,\, i\rangle) \preceq_{x;y} f(\langle e',\, i'\rangle)$, proving its monotonicity. Therefore, by Definition 3.2.6, $f \colon x \parallel y \to x; y$ is a monotonic bijective morphism, proving the claim. □

We have given the detailed proof of the previous proposition because it is a template for more complicated ones, and it illustrates how the concept of monotonic bijective morphisms by Ésik lends itself for constructive proofs about *inequalities* between partial strings, thereby generalizing the idea behind the constructive proofs about isomorphisms we have already seen. This way we can carry forward the simplicity and ingenuity of the pomset model to inequational reasoning. However, this added flexibility also means that proofs about partial strings may end up sometimes more combinatorial. The next example gives a first glimpse at why this may be more suitable for machines than humans.

Proposition 3.2.20 (Monotonicity)**.** *For all $x, y, z \in \mathsf{P}$, if $x \sqsubseteq y$, then $x \bowtie z \sqsubseteq y \bowtie z$ and $z \bowtie x \sqsubseteq z \bowtie y$.*

Proof. Assume $x \sqsubseteq y$. Show $x \parallel z \sqsubseteq y \parallel z$. Let $g\colon y \to x$ be a monotonic bijective morphism as a witness for the assumption. Let $f\colon E_{y\parallel z} \to E_{x\parallel z}$ be a function such that, for all $e \in E_y \cup E_z$ and $i \in \mathbb{B}$,

$$f(\langle e,\ i\rangle) = \begin{cases} \langle g(e),\ 0\rangle & \text{if } i = 0 \\ \langle e,\ 1\rangle & \text{if } i = 1 \end{cases}.$$

Clearly f is bijective and it preserves labels. Let $e' \in E_y \cup E_z$ and $i' \in \mathbb{B}$. Assume $\langle e,\ i\rangle \preceq_{y\parallel z} \langle e,\ i'\rangle$. By Definition 3.2.4 and the last assumption, there are only two cases to consider: either $i = i' = 0$ or $i = i' = 1$. If $i = i' = 0$, then $e \preceq_y e'$ and $\langle g(e),\ 0\rangle = f(\langle e,\ 0\rangle) \preceq_{x\parallel z} f(\langle e',\ 0\rangle) = \langle g(e'),\ 0\rangle$ because $g(e) \preceq_x g(e')$; otherwise, $\langle e,\ 1\rangle = f(\langle e,\ 1\rangle) \preceq_{x\parallel z} f(\langle e',\ 1\rangle) = \langle e',\ 1\rangle$ because $e \preceq_z e'$, proving that $x \sqsubseteq y$ implies $x \parallel z \sqsubseteq y \parallel z$. By $\parallel$-commutativity, if $x \sqsubseteq y$, then $z \parallel x \sqsubseteq z \parallel y$. The proof for ;-monotonicity is similar except that there are three cases to consider. □

In addition to being monotonic, both sequential and concurrent composition are associative up to isomorphism.

Proposition 3.2.21 (Associativity)**.** *For all* $x, y, z \in \mathsf{P}$, $(x \bowtie y) \bowtie z \cong x \bowtie (y \bowtie z)$.

Proof. Let $f\colon E_{(x\parallel y)\parallel z} \to E_{x\parallel(y\parallel z)}$ be a function such that, for all $e \in E_{(x\parallel y)\parallel z}$,

$$f(e) = \begin{cases} \langle\langle e',\ 0\rangle,\ 0\rangle & \text{if } \exists e' \in E_x\colon \langle e',\ 0\rangle = e \\ \langle\langle e',\ 1\rangle,\ 0\rangle & \text{if } \exists e' \in E_y\colon \langle\langle e',\ 0\rangle,\ 1\rangle = e \\ \langle e',\ 1\rangle & \text{if } \exists e' \in E_z\colon \langle\langle e',\ 1\rangle,\ 1\rangle = e \end{cases}.$$

We leave it as an exercise to the reader to verify that f is a partial string isomorphism, proving $\parallel$-associativity. Similarly for ;-associativity. □

For the theory of concurrency, we have already mentioned in Section 3.1 that the exchange law is important because it shows how concurrent and sequential composition can be interchanged. Operationally, it could be seen as a divide-and-conquer mechanism by which concurrent composition may be sequentially implemented on a machine. It follows a constructive proof of the exchange law for partial strings.

Proposition 3.2.22. *For all* $u, v, x, y \in \mathsf{P}$, $(u \parallel v);(x \parallel y) \sqsubseteq (u;x) \parallel (v;y)$.

Proof. Let $f\colon E_{(u;x)\parallel(v;y)} \to E_{(u\parallel v);(x\parallel y)}$ such that $f(\langle\langle e,\ i\rangle,\ j\rangle) \triangleq \langle\langle e,\ j\rangle,\ i\rangle$ for all events $e \in E_u \cup E_v \cup E_x \cup E_y$ and $i, j \in \mathbb{B}$. Clearly f is bijective. Moreover, the

following equalities hold by Definition 3.2.4 of the labelling function:

$$\begin{aligned}
\alpha_{(u;x)\|(v;y)}(\langle\langle e,\ i\rangle,\ j\rangle) &= \begin{cases} \alpha_{u;x}(\langle e,\ i\rangle) & \text{if } j = 0 \\ \alpha_{v;y}(\langle e,\ i\rangle) & \text{if } j = 1 \end{cases} && \{\text{Definition of } \|\} \\
&= \begin{cases} \alpha_u(e) & \text{if } i = 0 \text{ and } j = 0 \\ \alpha_v(e) & \text{if } i = 0 \text{ and } j = 1 \\ \alpha_x(e) & \text{if } i = 1 \text{ and } j = 0 \\ \alpha_y(e) & \text{if } i = 1 \text{ and } j = 1 \end{cases} && \{\text{Definition of } ;\} \\
&= \begin{cases} \alpha_{u\|v}(\langle e,\ j\rangle) & \text{if } i = 0 \\ \alpha_{x\|y}(\langle e,\ j\rangle) & \text{if } i = 1 \end{cases} && \{\text{Definition of } \|\} \\
&= \alpha_{(u\|v);(x\|y)}(\langle\langle e,\ j\rangle,\ i\rangle) && \{\text{Definition of } ;\} \\
&= \alpha_{(u\|v);(x\|y)}(f(\langle\langle e,\ i\rangle,\ j\rangle)). && \{\text{Definition of } f\}
\end{aligned}$$

In short, f preserves the labelling of events. Let $e' \in E_u \cup E_v \cup E_x \cup E_y$ and $i', j' \in \mathbb{B}$. Assume $\langle\langle e,\ i\rangle,\ j\rangle \preceq_{(u;x)\|(v;y)} \langle\langle e',\ i'\rangle,\ j'\rangle$. Show $f(\langle\langle e,\ i\rangle,\ j\rangle) \preceq_{(u\|v);(x\|y)} f(\langle\langle e',\ i'\rangle,\ j'\rangle)$. By definition of f, it suffices to show

$$\langle\langle e,\ j\rangle,\ i\rangle) \preceq_{(u\|v);(x\|y)} \langle\langle e',\ j'\rangle,\ i'\rangle.$$

By assumption and Definition 3.2.4 of concurrent composition, ($j = j' = 0$ and $\langle e,\ i\rangle \preceq_{(u;x)} \langle e',\ i'\rangle$) or ($j = j' = 1$ and $\langle e,\ i\rangle \preceq_{(v;y)} \langle e',\ i'\rangle$). By Definition 3.2.4 of sequential composition, it follows

$$\begin{aligned}
&\Big(j = j' = 0 \text{ and } \big(i < i' \text{ or } (i = i' = 0 \text{ and } e \preceq_u e') \text{ or } (i = i' = 1 \text{ and } e \preceq_x e')\big)\Big) \text{ or} \\
&\quad \Big(j = j' = 1 \text{ and } \big(i < i' \text{ or } (i = i' = 0 \text{ and } e \preceq_v e') \text{ or } (i = i' = 1 \text{ and } e \preceq_y e')\big)\Big).
\end{aligned}$$

From propositional logic follows

$$\begin{aligned}
&i < i' \text{ or } \Big(i = i' = 0 \text{ and } \big((j = j' = 0 \text{ and } e \preceq_u e') \text{ or } (j = j' = 1 \text{ and } e \preceq_v e')\big)\Big) \text{ or} \\
&\quad \Big(i = i' = 1 \text{ and } \big((j = j' = 0 \text{ and } e \preceq_x e') \text{ or } (j = j' = 1 \text{ and } e \preceq_y e')\big)\Big).
\end{aligned}$$

By Definition 3.2.4 of concurrent composition, $i < i'$ or $\langle e,\ j\rangle \preceq_{u\|v} \langle e',\ j'\rangle$ or

$\langle e,\ j\rangle \preceq_{x\parallel y} \langle e',\ j'\rangle$. Thus, by definition of sequential composition,

$$\langle\langle e,\ j\rangle,\ i\rangle \preceq_{(u\parallel v);(x\parallel y)} \langle\langle e',\ j'\rangle,\ i'\rangle.$$

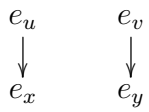
Therefore $f(\langle\langle e,\ i\rangle,\ j\rangle) \preceq_{(u\parallel v);(x\parallel y)} f(\langle\langle e',\ i'\rangle,\ j'\rangle)$ by definition of f. Therefore $f\colon (u;x) \parallel (v;y) \to (u \parallel v);(x \parallel y)$ is a monotonic bijective morphism, proving the claim by Definition 3.2.6. □

The exchange law is originally postulated in Gischer's thesis where it is called "subsumption axiom" [Gis85, p. 22]. Here, we prove that Gischer's exchange law holds for partial strings with respect to the refinement order $\sqsubseteq$. It is also interesting to compare the previous constructive proof with Ésik's argument why the exchange law is morally true [É02, Proposition 3.7]. In the context of formal verification tools, we prefer the previous construction because it retains more of the computational aspect of the problem.

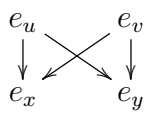
While it is not difficult to convince ourselves that the exchange law for partial strings is not an isomorphism, it is nevertheless worthwhile to do so because if they were isomorphic then the Eckmann-Hilton argument about any two monoid structures would imply that sequential and concurrent partial string operators coincide — something we usually would not want because it would conflate concepts that have typically different program semantics.

Proposition 3.2.23. *The exchange law for partial strings is not an isomorphism.*

Proof. Let u, v, x and y be partial strings that only consist of a single event e_u, e_v, e_x and e_y, respectively. Then $(u;x) \parallel (v;y)$ is the following partial string:



In contrast to $(u;x) \parallel (v;y)$, the partial string $(u \parallel v);(x \parallel y)$ looks as follows:



It is now easy to see that $(u \parallel v);(x \parallel y) \ncong (u;x) \parallel (v;y)$. □

The following corollary is directly Lemma 6.8 in [HMSW11]:

Corollary 3.2.24. *For all $x, y, z \in \mathsf{P}$, the following inequalities hold:*

- $(x \parallel y); z \sqsubseteq x \parallel (y; z)$,
- $x; (y \parallel z) \sqsubseteq (x; y) \parallel z$.

Proof. By Proposition 3.2.11 and 3.2.22, $(x \parallel y); z \cong (x \parallel y); (\bot \parallel z) \sqsubseteq (x; \bot) \parallel (y; z) \cong x \parallel (y; z)$. An analogous argument proves $x; y \parallel z \sqsubseteq x; (y \parallel z)$. □

It is not difficult to see that Corollary 3.2.24 implies Proposition 3.2.19. We have nevertheless given the direct proof of the latter because it served as an exemplar for other proofs about the refinement ordering between partial strings in terms of monotonic bijective morphisms. We next consider two natural least fixed point operators on the family of finite partial strings.

3.3 Program least fixed point reduction

So far, we have considered individual partial strings. This is about to change as we study programs where a single partial string is not enough to model mutually exclusive (non-deterministic) control flow. To see this, consider the simple (possibly sequential) system `if * then P else Q` where `*` denotes non-deterministic choice. If the semantics of a program was a single partial string, then we need to find exactly one partial string that represents the fact that `P` executes or `Q` executes, but never both. To model this kind of non-deterministic choice, we use the Hoare powerdomain construction, lifting sequential and concurrent composition operators to *sets* of partial strings.[2] Recall that each of these sets is downward closed with respect to the partial string refinement ordering from Definition 3.2.12 because we want to capture all possible ways events are ordered. By using the downward closure, program refinement coincides with familiar set inclusion, and we can easily define the familiar Kleene star operators. One of the main limitations of our Hoare powerdomain is that it cannot account for moments of choice. For example, consider the following two event structures where the wavy line represents the conflict relation [NPW81]:

Note that the set of partial strings for both event structures is the same [Res99].

[2]Section 3.1 explains the reason for choosing the Hoare powerdomain.

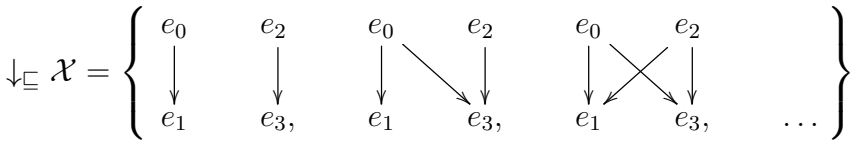


Figure 3.5: Downward-closure of the set $\mathcal{X} = \{x\}$ where x denotes the finite partial string in Figure 3.1

Definition 3.3.1 (Program)**.** A **program** is a downward-closed set of finite partial strings with respect to $\sqsubseteq$; equivalently $\mathcal{X} \subseteq \mathsf{P}_f$ is a program whenever $\downarrow_{\sqsubseteq} \mathcal{X} = \mathcal{X}$ where $\downarrow_{\sqsubseteq} \mathcal{X} \triangleq \{y \in \mathsf{P}_f \mid \exists x \in \mathcal{X} \colon y \sqsubseteq x\}$. Denote with $\mathbb{P}$ the family of all programs.

Since we only consider systems that terminate, each partial string x in a program $\mathcal{X}$ is finite. We reemphasize that the downward closure of such a set $\mathcal{X}$ can be thought of as an over-approximation of all possible happens-before relations in a concurrent system whose instructions are ordered according to the partial strings in $\mathcal{X}$. In Chapter 4, we make the downward closure of partial strings more precise to model a certain kind of relaxed sequential consistency.

Example 3.3.2. Recall that N-shaped partial strings cannot be constructed under any labelling using sequential and concurrent composition, i.e. it is not a series-parallel pomset [Pra86]. Yet, by downward-closure of programs, such partial strings are included in the over-approximation of all the happens-before relations exhibited by a concurrent system. In particular, according to Example 3.2.13, the downward-closure of the set containing the partial string in Figure 3.1 includes (among many others) the N-shaped partial string shown on the right in Figure 3.3. This is illustrated in Figure 3.5. In fact, we shall see in Chapter 4 that this particular N-shaped partial string corresponds to a data race in the concurrent system shown in Figure 4.1.

It is standard to define $0 \triangleq \emptyset$ and $1 \triangleq \{\bot\}$ where $\bot$ is the (unique) empty partial string. Clearly 0 and 1 form programs in the sense of Definition 3.3.1.

Since $\mathbb{P}$ is the family of of $\sqsubseteq$-downward-closed sets of finite partial strings, $\mathbb{P}$ clearly forms a complete lattice that is ordered by subset inclusion, $\subseteq$, where meet, denoted by $\cap$, corresponds to set intersection and join, denoted by $\cup$, is set union with the empty set as bottom, 0, and the whole of P_f as top.

Definition 3.3.3 (Program refinement)**.** $\subseteq$ on $\mathbb{P}$ is called **program refinement**.

In the theory of denotational semantics, the complete lattice of programs is called a *Hoare powerdomain* where the ordering $\mathcal{P} \subseteq \mathcal{Q}$ for programs $\mathcal{P}$ and $\mathcal{Q}$ says that $\mathcal{P}$

is more deterministic than $\mathcal{Q}$, or that $\mathcal{P}$ *refines* $\mathcal{Q}$. Semantically, the operator $\mathcal{P} \cup \mathcal{Q}$ could therefore be seen as the non-deterministic choice of either $\mathcal{P}$ or $\mathcal{Q}$. It follows that the equality of programs $\mathcal{P}$ and $\mathcal{Q}$ is the same as their subset inclusion both ways, i.e. $\mathcal{P} = \mathcal{Q}$ is equivalent to $\mathcal{P} \subseteq \mathcal{Q}$ and $\mathcal{Q} \subseteq \mathcal{P}$. This means the following equivalence holds:

Proposition 3.3.4. *For all* $\mathcal{X}, \mathcal{Y} \in \mathbb{P}$, $\mathcal{X} \subseteq \mathcal{Y}$ *exactly if* $\forall x \in \mathcal{X} \colon \exists y \in \mathcal{Y} \colon x \sqsubseteq y$.

Proof. Assume $\mathcal{X} \subseteq \mathcal{Y}$. Let $x \in \mathcal{X}$. By assumption, $x \in \mathcal{Y}$. By reflexivity of $\sqsubseteq$ (Proposition 3.2.15), $x \sqsubseteq x$. Thus, $\forall x \in \mathcal{X} \colon \exists y \in \mathcal{Y} \colon x \sqsubseteq y$.

Conversely, assume $\forall x \in \mathcal{X} \colon \exists y \in \mathcal{Y} \colon x \sqsubseteq y$. Let $x \in \mathcal{X}$. By assumption, there exists $y \in \mathcal{Y}$ such that $x \sqsubseteq y$. Since programs are downward-closed sets with respect to $\sqsubseteq$ (Definition 3.3.1), $x \in \mathcal{Y}$. Thus, $\mathcal{X} \subseteq \mathcal{Y}$. □

For the next theorem, we lift the two partial string operators (Definition 3.2.4) to programs in the standard way:

Definition 3.3.5 (Lifted bow tie)**.** For every program $\mathcal{X}$ and $\mathcal{Y}$ in $\mathbb{P}$, and partial string operator $\bowtie$, define $\mathcal{X} \bowtie \mathcal{Y} \triangleq \downarrow_{\sqsubseteq} \{x \bowtie y \mid x \in \mathcal{X} \text{ and } y \in \mathcal{Y}\}$ where $\mathcal{X} \parallel \mathcal{Y}$ and $\mathcal{X}; \mathcal{Y}$ are called **concurrent** and **sequential program composition**, respectively.

The meaning of program composition operators derive from their counterpart in the partial string model from Section 3.2. As in the case of partial string operators, we use the 'bow tie' operator, denoted by $\bowtie$, as placeholder for either concurrent or sequential program composition (recall Definition 3.2.10).

By denoting programs as sets of partial strings, we can now define Kleene star operators $(-)^{\parallel}$ and $(-)^{;}$ for iterative concurrent and sequential program composition, respectively, as least fixed points (μ) using the non-deterministic choice operator ($\cup$) as the binary join operator. We remark that this is fundamentally different from the pomsets recursion operators in ultra-metric spaces [dBW90]. The next theorem could be then summarized as saying that the structure of programs, written $\mathfrak{S}$, is a partial-order model of an algebraic concurrency semantics that satisfies the CKA laws [HMSW11]. Since CKA is an exemplar of the universal laws of programming [HvS12b], the rest of this chapter is based on our partial-order model of CKA.

Theorem 3.3.6. *The structure* $\mathfrak{S} = \langle \mathbb{P}, \subseteq, \cup, 0, 1, ;, \parallel \rangle$ *is a complete lattice, ordered by subset inclusion (i.e.* $\mathcal{X} \subseteq \mathcal{Y}$ *exactly if* $\mathcal{X} \cup \mathcal{Y} = \mathcal{Y}$*), such that* $\parallel$ *and* $;$ *form unital*

quantales over $\cup$ *where* $\mathfrak{S}$ *satisfies the following:*

$$(\mathcal{U} \parallel \mathcal{V}); (\mathcal{X} \parallel \mathcal{Y}) \subseteq (\mathcal{U}; \mathcal{X}) \parallel (\mathcal{V}; \mathcal{Y}) \qquad \mathcal{X} \cup (\mathcal{Y} \cup \mathcal{Z}) = (\mathcal{X} \cup \mathcal{Y}) \cup \mathcal{Z}$$
$$\mathcal{X} \cup \mathcal{X} = \mathcal{X} \qquad \mathcal{X} \cup 0 = 0 \cup \mathcal{X} = \mathcal{X}$$
$$\mathcal{X} \cup \mathcal{Y} = \mathcal{Y} \cup \mathcal{X} \qquad \mathcal{X} \parallel \mathcal{Y} = \mathcal{Y} \parallel \mathcal{X}$$
$$\mathcal{X} \parallel 1 = 1 \parallel \mathcal{X} = \mathcal{X} \qquad \mathcal{X}; 1 = 1; \mathcal{X} = \mathcal{X}$$
$$\mathcal{X} \parallel 0 = 0 \parallel \mathcal{X} = 0 \qquad \mathcal{X}; 0 = 0; \mathcal{X} = 0$$
$$\mathcal{X} \parallel (\mathcal{Y} \cup \mathcal{Z}) = (\mathcal{X} \parallel \mathcal{Y}) \cup (\mathcal{X} \parallel \mathcal{Z}) \qquad \mathcal{X}; (\mathcal{Y} \cup \mathcal{Z}) = (\mathcal{X}; \mathcal{Y}) \cup (\mathcal{X}; \mathcal{Z})$$
$$(\mathcal{X} \cup \mathcal{Y}) \parallel \mathcal{Z} = (\mathcal{X} \parallel \mathcal{Z}) \cup (\mathcal{Y} \parallel \mathcal{Z}) \qquad (\mathcal{X} \cup \mathcal{Y}); \mathcal{Z} = (\mathcal{X}; \mathcal{Z}) \cup (\mathcal{Y}; \mathcal{Z})$$
$$\mathcal{X} \parallel (\mathcal{Y} \parallel \mathcal{Z}) = (\mathcal{X} \parallel \mathcal{Y}) \parallel \mathcal{Z} \qquad \mathcal{X}; (\mathcal{Y}; \mathcal{Z}) = (\mathcal{X}; \mathcal{Y}); \mathcal{Z}$$
$$\mathcal{P}^{\parallel} = \mu\mathcal{X}.1 \cup (\mathcal{P} \parallel \mathcal{X}) \qquad \mathcal{P}^{;} = \mu\mathcal{X}.1 \cup (\mathcal{P}; \mathcal{X}).$$

Proof. The details of the proof can be found in Appendix D. □

By Theorem 3.3.6, it therefore makes sense to call 1 in structure $\mathfrak{S}$ the **$\bowtie$-identity program** where $\bowtie$ is a placeholder for either ; or $\parallel$.

Finally, we are ready to devote our attention to the least fixed point operators $(-)^{;}$ and $(-)^{\parallel}$. Henceforth, we shall denote these by $(-)^{\bowtie}$. We show that under a certain finiteness condition (Definition 3.3.9) the program refinement problem $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$ can be reduced to a bounded number of program refinement problems without least fixed points (Theorem 3.3.15). To prove this, we start by inductively defining the notion of iteratively composing a program with itself under $\bowtie$.

Definition 3.3.7 (n-iterated-$\bowtie$-program-composition)**.** Let $\mathbb{N}_0 \triangleq \mathbb{N} \cup \{0\}$ be the set of **non-negative integers**. For every program $\mathcal{P}$ in $\mathbb{P}$ and non-negative integer n in $\mathbb{N}_0$, let $\mathcal{P}^{0\cdot\bowtie} \triangleq 1 = \{\bot\}$ be the $\bowtie$-identity program and $\mathcal{P}^{(n+1)\cdot\bowtie} \triangleq \mathcal{P} \bowtie \mathcal{P}^{n\cdot\bowtie}$.

Clearly $(-)^{\bowtie}$ is the limit of its approximations in the following sense:

Proposition 3.3.8. *For every program* $\mathcal{P}$ *in* $\mathbb{P}$, $\mathcal{P}^{\bowtie} = \bigcup_{n\geq 0} \mathcal{P}^{n\cdot\bowtie}$.

Proof. Let $F_{\mathcal{P}} \colon \mathbb{P} \to \mathbb{P}$ such that, for every program $\mathcal{X} \in \mathbb{P}$, $F_{\mathcal{P}}(\mathcal{X}) = 1 \cup (\mathcal{P} \bowtie \mathcal{X})$. Since $\bowtie$ distributes through arbitrary unions, $F_{\mathcal{P}}$ is also universally distributive and hence continuous, proving the claim by Kleene's theorem. □

Definition 3.3.9 (Elementary program)**.** A program $\mathcal{P}$ in $\mathbb{P}$ is called **elementary** if $\mathcal{P}$ is the downward-closed set with respect to $\sqsubseteq$ of some finite and nonempty set $\mathcal{Q}$ of finite partial strings such that the empty partial string is not in $\mathcal{Q}$, i.e. $\mathcal{P} = {\downarrow_{\sqsubseteq}}\, \mathcal{Q}$ and $\bot \notin \mathcal{Q}$ where $\emptyset \neq \mathcal{Q} \subseteq \mathsf{P}_f$ is finite. Let $\mathbb{P}_\ell$ be the set of elementary programs.

An elementary program could be therefore seen as a machine-representable program generated from a finite and nonempty set of finite partial strings. This finiteness restriction makes the notion of elementary programs a suitable candidate for the study of decision procedures. To make this precise, we define the following unary partial string operator:

Definition 3.3.10 (n-repeated-$\bowtie$ partial string operator)**.** For every non-negative integer n in $\mathbb{N}_0$, let $x^{0\cdot\bowtie} \triangleq \bot$ be the empty partial string and $x^{(n+1)\cdot\bowtie} \triangleq x \bowtie x^{n\cdot\bowtie}$.

Intuitively, $p^{n\cdot\bowtie}$ is a partial string that consists of n copies of a partial string p, each combined by the partial string operator $\bowtie$. This is formalized as follows:

Proposition 3.3.11. *Let $n \in \mathbb{N}_0$ be a non-negative integer. Define $[0] \triangleq \emptyset$ to be the empty set and $[n+1] \triangleq \{1, \ldots, n+1\}$. For every partial string x, $x^{n\cdot\bowtie}$ is isomorphic to $y = \langle E_y, \alpha_y, \preceq_y \rangle$ where $E_y \triangleq E_x \times [n]$ such that, for all $e, e' \in E_x$ and $i, i' \in [n]$, the following holds:*

- *if '$\bowtie$' is '$\|$', then $\langle e,\ i\rangle \preceq_y \langle e',\ i'\rangle$ exactly if $i = i'$ and $e \preceq_x e'$,*
- *if '$\bowtie$' is ';', then $\langle e,\ i\rangle \preceq_y \langle e',\ i'\rangle$ exactly if $i < i'$ or ($i = i'$ and $e \preceq_x e'$),*
- *$\alpha_y(\langle e,\ i\rangle) = \alpha_x(e)$.*

Definition 3.3.12 (Partial string size)**.** The **size** of a finite partial string p, denoted by $|p|$, is the cardinality of its event set E_p.

For example, the partial string in Figure 3.1 has size four. It is obvious that the size of finite partial strings is non-decreasing under the n-repeated-$\bowtie$ partial string operator from Definition 3.3.10 whenever $0 < n$. This simple fact is important for the next step towards our least fixed point reduction result in Theorem 3.3.15:

Proposition 3.3.13 (Elementary least fixed point pre-reduction)**.** *For all elementary programs $\mathcal{X}$ and $\mathcal{Y}$ in $\mathbb{P}_\ell$, $\mathcal{X} \subseteq \bigcup_{n \geq k \geq 0} \mathcal{Y}^{k\cdot\bowtie}$ where $n = \left\lfloor \frac{\ell_\mathcal{X}}{\ell_\mathcal{Y}} \right\rfloor$ such that $\ell_\mathcal{X} \triangleq \mathsf{max}\{|x| \mid x \in \mathcal{X}\}$ and $\ell_\mathcal{Y} \triangleq \mathsf{min}\{|y| \mid y \in \mathcal{Y}\}$ is the size of the largest and smallest partial strings in $\mathcal{X}$ and $\mathcal{Y}$, respectively.*

Proof. Assume $\mathcal{X} \subseteq \mathcal{Y}^{\bowtie}$. Let $x \in \mathsf{P}_f$ be a finite partial string. We can assume $x \in \mathcal{X}$ because $\mathcal{X} \neq 0$. By assumption, $x \in \mathcal{Y}^{\bowtie}$. By Proposition 3.3.8, there exists $k \in \mathbb{N}_0$ such that $x \in \mathcal{Y}^{k\cdot\bowtie}$. Fix k to be the smallest such non-negative integer. Show $k \leq \left\lfloor \frac{\ell_\mathcal{X}}{\ell_\mathcal{Y}} \right\rfloor$ (the fraction is well-defined because $\mathcal{X}$ and $\mathcal{Y}$ are nonempty and $1 \notin \mathcal{Y}$). By downward closure and definition of $\sqsubseteq$ in terms of a one-to-one correspondence, it

suffices to consider that x is one of a (not necessarily unique) longest partial strings in $\mathcal{X}$, i.e. $|x'| \leq |x|$ for all $x' \in \mathcal{X}$; equivalently, $|x| = \ell_{\mathcal{X}}$. Since $\bot \notin \mathcal{X}, \mathcal{Y}$ and the size of partial strings in a program can never decrease under the k-iterated program composition operator $\bowtie$ when $0 < k$, it suffices to consider the case $x \sqsubseteq y^{k\cdot\bowtie}$ for some shortest partial string y in $\mathcal{Y}$. Since $E_{y^{k\cdot\bowtie}}$ is the Cartesian product of E_y and $[k]$, it follows $|x| = k \cdot |y|$. Since $|x| \leq \ell_{\mathcal{X}}$ and $\ell_{\mathcal{Y}} \leq |y|$, $k \leq \left\lfloor \frac{\ell_{\mathcal{X}}}{\ell_{\mathcal{Y}}} \right\rfloor$. By definition $n = \left\lfloor \frac{\ell_{\mathcal{X}}}{\ell_{\mathcal{Y}}} \right\rfloor$, proving $x \in \bigcup_{n \geq k \geq 0} \mathcal{Y}^{k\cdot\bowtie}$. □

Equivalently, if there exists a partial string x in $\mathcal{X}$ such that $x \notin \mathcal{Y}^{k\cdot\bowtie}$ for all non-negative integers k between zero and $\left\lfloor \frac{\ell_{\mathcal{X}}}{\ell_{\mathcal{Y}}} \right\rfloor$, then $\mathcal{X} \not\subseteq \mathcal{Y}^{\bowtie}$. Since we are interested in decision procedures for program refinement checking, we need to show that the converse of Proposition 3.3.13 also holds. Towards this end, we prove the following left $(-)^{\bowtie}$ elimination rule:

Proposition 3.3.14. *For every program $\mathcal{X}$ and $\mathcal{Y}$ in $\mathbb{P}$, $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$ exactly if $\mathcal{X} \subseteq \mathcal{Y}^{\bowtie}$.*

Proof. Assume $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$. By Proposition 3.3.8, $\mathcal{X} \subseteq \mathcal{X}^{\bowtie}$. By transitivity of $\subseteq$ in $\mathbb{P}$, $\mathcal{X} \subseteq \mathcal{Y}^{\bowtie}$. Conversely, assume $\mathcal{X} \subseteq \mathcal{Y}^{\bowtie}$. Let $i, j \in \mathbb{N}_0$. By induction on i, $\mathcal{X}^{i\cdot\bowtie} \bowtie \mathcal{X}^{j\cdot\bowtie} = \mathcal{X}^{(i+j)\cdot\bowtie}$. Thus, by Proposition 3.3.8 and distributivity of $\bowtie$ over least upper bounds in $\mathbb{P}$, $\mathcal{X}^{\bowtie} \bowtie \mathcal{X}^{\bowtie} = \mathcal{X}^{\bowtie}$, i.e. $(-)^{\bowtie}$ is idempotent. This, in turn, implies that $(-)^{\bowtie}$ is a closure operator. Therefore, by monotonicity, $\mathcal{X}^{\bowtie} \subseteq (\mathcal{Y}^{\bowtie})^{\bowtie} = \mathcal{Y}^{\bowtie}$, proving that $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$ is equivalent to $\mathcal{X} \subseteq \mathcal{Y}^{\bowtie}$. □

We use Proposition 3.3.14 to prove the main result of this section.

Theorem 3.3.15 (Elementary least fixed point reduction)**.** *For all elementary programs $\mathcal{X}$ and $\mathcal{Y}$ in $\mathbb{P}_\ell$, $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$ is equivalent to $\mathcal{X} \subseteq \bigcup_{n \geq k \geq 0} \mathcal{Y}^{k\cdot\bowtie}$ where $n = \left\lfloor \frac{\ell_{\mathcal{X}}}{\ell_{\mathcal{Y}}} \right\rfloor$ such that $\ell_{\mathcal{X}} \triangleq \mathsf{max}\,\{|x| \mid x \in \mathcal{X}\}$ and $\ell_{\mathcal{Y}} \triangleq \mathsf{min}\,\{|y| \mid y \in \mathcal{Y}\}$ is the size of the largest and smallest partial strings in $\mathcal{X}$ and $\mathcal{Y}$, respectively.*

Proof. By Proposition 3.3.14, it remains to show that $\mathcal{X} \subseteq \mathcal{Y}^{\bowtie}$ is equivalent to $\mathcal{X} \subseteq \bigcup_{n \geq k \geq 0} \mathcal{Y}^{k\cdot\bowtie}$ where $n = \left\lfloor \frac{\ell_{\mathcal{X}}}{\ell_{\mathcal{Y}}} \right\rfloor$. The forward and backward implication follow from Proposition 3.3.13 and 3.3.8, respectively. □

From Theorem 3.3.15 follows immediately that $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$ is decidable for all elementary programs $\mathcal{X}$ and $\mathcal{Y}$ in $\mathbb{P}_\ell$ because there exists an algorithm that could iteratively make $O\left(|\mathcal{X}| \times |\mathcal{Y}|^n\right)$ calls to another decision procedure to check whether $x \sqsubseteq y$ for all $x \in \mathcal{X}$ and $y \in \mathcal{Y}^{k\cdot\bowtie}$ where $n \geq k \geq 0$. However, by Proposition 3.2.18, each iteration in such an algorithm would have to solve an NP-complete subproblem. Alternatively, it would be possible to avoid the iterative call to the NP-complete

decision procedure by symbolically encoding all finite partial strings in the right-hand side of the program refinement check. However, there are an exponential of these partial strings. In either case, the high complexity of the algorithm is expected since the pomset language containment problem is Π_2^p-complete [FKL93].

Corollary 3.3.16. *For all elementary programs $\mathcal{X}$ and $\mathcal{Y}$ in $\mathbb{P}$, if $|x| = |y|$ for all $x \in \mathcal{X}$ and $y \in \mathcal{Y}$, then $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$ is equivalent to $\mathcal{X} \subseteq \mathcal{Y}$.*

We next move on to a reference implementation of the previous decision procedure in order to quantify the effects of the underlying NP-complete decision procedure.

3.4 Implementation and experiments

We carefully defined partial strings and the concept of partial string refinement with the goal of leveraging SAT/SMT solvers. To prove this point and answer the question what difference the choice of solver can make in performance, we give the technical details on how to encode finite partial strings using quantifier-free first-order logic of equality and uninterpreted functions. Since this is one of the most basic theories supported by SMT solvers, our reference implementation is very portable. Our encoding of finite partial strings addresses the following decision problems:

Partial string refinement (PSR)

INPUT: Let x and y be finite partial strings in P_f.

QUESTION: Is $x \sqsubseteq y$?

In other words, the PSR problem asks whether x refines y. By Definition 3.2.6, this problem is equivalent to checking for the existence of a monotonic bijective morphism from y to x, and is NP-complete by Proposition 3.2.18.

The second problem, which is decidable by Theorem 3.3.15, is about a fragment of CKA that concerns elementary programs and least fixed points:

Elementary program refinement-$\bowtie$ (EPR$^{\bowtie}$)

INPUT: Let $\mathcal{X}$ and $\mathcal{Y}$ be elementary programs in $\mathbb{P}_\ell$.

QUESTION: Is $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$?

For both the PSR and EPR$^{\bowtie}$ problem, we give a reference implementation that we experimentally evaluate using MathSAT5 and Z3.

Algorithm 1 An SMT-based partial-order encoding of the PSR problem.

Require: Let $\langle E_x, \alpha_x, \preceq_x \rangle$ and $\langle E_y, \alpha_y, \preceq_y \rangle$ be nonempty finite partial strings.
Require: Let n be a natural number such that $n = |E_x| = |E_y|$.
Require: Without loss of generality, assume $E_x = E_y = \mathbb{N}_n$ where $\mathbb{N}_n = \{0, \ldots, n\}$.
Require: For all events $e \in \mathbb{N}_n$, the labels $\alpha_x(e)$ and $\alpha_y(e)$ are integers.

```
function CHECK_PSR(x, y)
    PRINT((set-logic QF_UFLIA))
    PRINT((declare-fun order (Int Int) Bool))
    PRINT((declare-fun event (Int) Int))
    PRINT((declare-fun label (Int) Int))
    for all events e ∈ ℕ_n do PRINT((= (label e) α_x(e)))      ▷ Encode labels of x
    for all events e, e' ∈ ℕ_n do                              ▷ Encode ⪯_x partial-order
        if e = e' then
            PRINT((not (order e e')))                          ▷ Irreflexivity
        else if e ≺_x e' then
            PRINT((and (order e e') (not (order e' e))))       ▷ Strictness
        else if e' ⊀_x e then
            PRINT((not (or (order e e') (order e' e))))        ▷ Incomparability
    for all events e, e' ∈ ℕ_n such that e ≺_y e' do           ▷ Encode strict monotonicity
        PRINT((order (event e) (event e')))
    PRINT((distinct {(event e) | e ∈ ℕ_n}))                    ▷ Encode injectivity
    onto ← (true)
    for all events e ∈ ℕ_n do                                  ▷ Encode surjectivity and labels of y
        onto ← (and onto (<= 0 (event e)) (<= (event e) n))
        PRINT((= (label (event e)) α_y(e)))
    PRINT(onto)
    return PRINT((check-sat))                                  ▷ Checker whether x refines y, i.e. x ⊑ y?
```

3.4.1 An SMT encoding of the PSR and EPR$^\times$ problem

We start by symbolically encoding finite partial strings in the quantifier-free first-order logic of linear integer arithmetic with uninterpreted functions (`QF_UFLIA`) as defined by the SMT-LIB standard [BFT15]. Alternatively, we could have used, say, the quantifier-free first-order logic of uninterpreted functions and fixed-size bit vectors (`QF_UFBV`). From a complexity theory perspective, either symbolic encoding is fully justified by Proposition 3.2.18.

Among the three algorithms that follow, Algorithm 1 is the most fundamental one because it directly interacts with an SMT solver. The input to Algorithm 1 comprises two finite partial strings x and y. Given such x and y, the algorithm outputs a machine-readable quantifier-free first-order logic formula in the SMT-LIB format [BFT15]. It is not difficult to see that this formula is satisfiable if and only if

Algorithm 2 Refinement check $\mathcal{X} \subseteq \mathcal{Y}$ for elementary programs.

```
function CHECK_EPR(𝒳, 𝒴)
    for all partial strings x ∈ 𝒳 do
        is_refine ← false
        for all partial strings y ∈ 𝒴 do
            if E_x = E_y and CHECK_PSR(x, y) then
                is_refine ← true                    ▷ See Algorithm 1
        if not is_refine then return false
    return true
```

x refines y, i.e. $x \sqsubseteq y$. Since it is trivially the case that $x \not\sqsubseteq y$ when the number of events in x and y is different, the algorithm assumes that E_x and E_y are equal. More specifically, it assumes $E_x = E_y = \mathbb{N}_n$ where n is the number of events in x and y, and $\mathbb{N}_n = \{0, \ldots, n\}$ is the set of n non-negative integers. Of course, this assumption can always be satisfied by simply renaming events if necessary. Again without loss of generality, the last pre-condition in Algorithm 1 means that the labels of events are integers.

We now lift the previous symbolic encoding to programs. Since programs are downward-closed sets of partial strings with respect to the partial string refinement order $\sqsubseteq$, $\mathcal{X} \subseteq \mathcal{Y}$ exactly if $\forall x \in \mathcal{X} \colon \exists y \in \mathcal{Y} \colon x \sqsubseteq y$ for all programs $\mathcal{X}$ and $\mathcal{Y}$ in $\mathbb{P}$, see Proposition 3.3.4. Algorithm 2 therefore repeatably calls Algorithm 1 for each pair of finite partial strings in two elementary programs $\mathcal{X}$ and $\mathcal{Y}$, proving that the algorithm is sound with respect to the partial string model of CKA.

Proposition 3.4.1 (Soundness)**.** *For every program $\mathcal{X}$ and $\mathcal{Y}$, if* CHECK_EPR$(\mathcal{X}, \mathcal{Y})$ *terminates, it gives a 'yes' answer exactly if $\mathcal{X} \subseteq \mathcal{Y}$ with respect to $\mathfrak{S}$ from Theorem 3.3.6.*

Of course, in the case $\mathcal{X}$ and $\mathcal{Y}$ are not elementary, Algorithm 2 may generally be only a semi-decision procedure because the set of partial strings could be infinite. This is where Theorem 3.3.15 comes in: it can reduce the problem of checking an infinite set of partial strings to checking a *finite* set of partial strings. The hypothesis of the theorem states that this reduction is possible when a program is elementary according to Definition 3.3.9. Algorithm 3 is an computational rendition of Theorem 3.3.15.

Our reference implementation of Algorithms 1, 2 and 3 leverages SMT Kit (see Appendix C). This keeps our implementation to around 1,000 lines of code (excluding approximately 1,500 lines of unit tests). We remark that our implementation directly invokes a given SMT solver through its programmatic interface (rather than the usual

textual SMT-LIB format). We deemed this important because it rules out file I/O as a possible interference in experiments in Subsection 3.4.2, especially given the high frequency of SMT solver queries.

For illustration purposes, it follows an example of a program refinement check that uses our symbolic implementation of partial strings and programs.

Example 3.4.2. Consider the following C++11 code that constructs two CKA programs `P` and `Q` using sequential and concurrent composition operators:

```
Program U{'u'}, V{'v'}, X{'x'}, Y{'y'};
Program P{((U | V) , (X | Y))};
Program Q{((U , X) | (V , Y))};
assert(lfp<',','>(P) <= lfp<',','>(Q));
```

The previous C++11 code relies on our `Program` class and the overloaded binary operators '`|`' and '`,`' for concurrent and sequential composition, respectively. The benefit of using operator overloading is that we do not have to write a parser; instead, we can leverage the C++ compiler. The line `P{((U | V), (X | Y))}` denotes a program $\mathcal{P} \triangleq (\mathcal{U} \parallel \mathcal{V}); (\mathcal{X} \parallel \mathcal{Y})$ where $\mathcal{U}, \mathcal{V}, \mathcal{X}$ and $\mathcal{Y}$ are elementary programs each constructed from of a single partial string which contains exactly one event labelled by `'u'`, `'v'`, `'x'` and `'y'`, respectively. We remark that the implementation also features the overloaded binary operator '`+`' for the non-deterministic choice of two programs. The least fixed point of a program `P` is represented by the return value of the template function `lfp` whose template parameter corresponds to the 'bow tie' placeholder ($\bowtie$) in Section 3.3. Note that the assertion in the example uses the overloaded binary relational operator '`<=`' to check program refinement. In this example, the assertion always holds because the partial string model of CKA satisfies the program exchange law according to Theorem 3.3.6.

We have optimized the way a `Program` object is composed with itself under the 'bowtie' operator. This optimization is implemented in the `LazyProgram` class shown in

Algorithm 3 Refinement check for the EPR$^{\bowtie}$ problem.

function CHECK_EPR$^{\bowtie}(\mathcal{X}, \mathcal{Y})$
  $max\_x \leftarrow \mathbf{max}\{\ |x| \mid x \in \mathcal{X}\ \}$ ▷ Length of longest partial string in $\mathcal{X}$
  $min\_y \leftarrow \mathbf{min}\{\ |y| \mid y \in \mathcal{Y}\ \}$ ▷ Length of shortest partial string in $\mathcal{Y}$
  $j \leftarrow max\_x / min\_y$ ▷ Integer division
  $\mathcal{K} \leftarrow 1$ ▷ Initialize to identity program
  for $0, \ldots, j$ **do** ▷ Iterate $j + 1$ times
    $\mathcal{K} \leftarrow \mathcal{K} \bowtie \mathcal{Y}$ ▷ Compose $\mathcal{K}$ with $\mathcal{Y}$
    if CHECK_EPR$(\mathcal{X}, \mathcal{K})$ **then return true** ▷ See Algorithm 2, $\mathcal{X}^{\bowtie} \subseteq \mathcal{Y}^{\bowtie}$
  return false ▷ After loop, $\mathcal{X}^{\bowtie} \not\subseteq \mathcal{Y}^{\bowtie}$

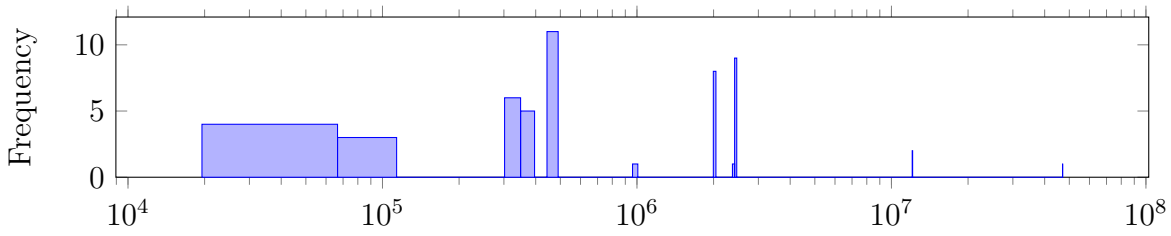


Figure 3.6: Log-scaled histogram of the number of loop iterations in Algorithm 3.

Listing 3.1. This class internally keeps a `Program` object subject to self-composition. Internally, `LazyProgram` uses a custom iterator of partial strings, `PartialStringIterator`, shown in Listing 3.2. The idea behind `PartialStringIterator` is to keep a variable-size array, `m_vector`, whose elements identify partial strings in the program undergoing self-composition. This allows us to lazily compose the partial strings in the program by iterating over all possible permutations of `m_vector`. More accurately, we use `m_vector` to iteratively compose the partial strings identified by `m_vector[i]` for all valid vector indexes `i`. Since this vector is shared between the `PartialStringIterator` and `Program` object for which the iterator is constructed, at most one iterator object can be instantiated at a given time for a particular `Program` object. This also explains why we have not implemented a more standard iterator interface such as `std::iterator<std::input_iterator_tag, const PartialString>` because this would require copies of possibly long vectors.

While simple, our reference implementation illustrates how theoretical concepts, such as partial strings and monotonic bijective morphisms, can be directly encoded into SMT-LIB formulas. We designed a low-memory footprint data structure for programs. We evaluatue our optimized implementation and run experiments using MathSAT5 and Z3.

3.4.2 Experiments

We evaluate our decision procedure for EPR$^{\bowtie}$. To eliminate any bias, our evaluation is carried out on a random data set that is obtained by uniform random generation of strings in a context-free language, e.g. [HC83]. For our purposes, we randomly generate the benchmarks from the following context-free grammar:

$\langle P \rangle$ ::= `U` | `V` | `W` | `X` | `Y` | `Z` | ($\langle P \rangle$ ‘`,`’ $\langle P \rangle$) | ($\langle P \rangle$ ‘`|`’ $\langle P \rangle$) | ($\langle P \rangle$ ‘`+`’ $\langle P \rangle$)

where `U` up to `Z` are `Program` objects instantiated with a single partial string. For example, `Program U{'u'}` instantiates a program in the sense of Definition 3.3.1 that consists of a single partial string that contains a single event which is labelled by `'u'`. Therefore,

```
  /// Raise ‘base‘ to the power of ‘exp‘, i.e. ‘base^exp’
  unsigned uint_pow(unsigned base, unsigned exp);

  template<Opchar opchar>
  class LazyProgram
  {
  private:
    // iteratively compose ‘*m_program_ptr’ under ‘opchar’
    const Program* m_program_ptr;

    // array of indexes into ‘m_program_ptr->partial_strings()’
    std::vector<unsigned> m_vector;

  public:
    LazyProgram(const Program& program_ref)
    : m_program_ptr{&program_ref}, // keep pointer to program
      m_vector{0U} // insert zero into the vector
    {}

    const Program& P() const noexcept
    {
      return *m_program_ptr;
    }

    /// Program size grows exponentially with every call to ‘extend()’
    unsigned size() const noexcept
    {
      return uint_pow(m_program_ptr->size(), m_vector.size());
    }

    /// Compose this program with itself
    void extend()
    {
      std::fill(m_vector.begin(), m_vector.end(), 0U);
      m_vector.push_back(zero);
    }

    /// cheap but at most one iterator can be used at a given time

    /// The iterator is in the same state the previous owner left it in.
    /// \see also PartialStringIterator<opchar>::reset()
    PartialStringIterator<opchar> partial_string_iterator() noexcept
    {
      return {m_program_ptr, m_vector};
    }
  };
}
```

Listing 3.1: Asymptotically, the n-repeated composition of a `Program` object requires exponential space in n. The purpose of ‘LazyProgram’ is to reduce this to an asymptotic linear space requirement.

```
/// Internally used by ‘LazyProgram<opchar>’
template<Opchar opchar>
class PartialStringIterator
{
private:
  const Program* m_program_ptr;
  std::vector<unsigned>& m_vector;

public:
  PartialStringIterator(const Program* program_ptr, std::vector<unsigned>& vector)
  : m_program_ptr{program_ptr}, m_vector(vector) {}

  bool has_next_partial_string() const noexcept
  {
    return m_program_ptr != nullptr and not m_vector.empty();
  }

  void reset()
  {
    std::fill(m_vector.begin(), m_vector.end(), 0);
  }

  /// \pre: ‘has_next_partial_string()’
  PartialString next_partial_string()
  {
    assert(has_next_partial_string());

    PartialString p{PartialString::empty()};
    for (unsigned i : m_vector)
    {
      assert(i < m_program_ptr->size());

      // compose ‘p’ with ‘i’th partial string in ‘*m_program_ptr’
      p = Eval<opchar>::bowtie(p, m_program_ptr->partial_strings().at(i));
    }

    // detect whether another partial string can be generated
    bool is_end = true;

    // similar to a ripple-carry adder that computes
    // ‘m_vector + 1’ modulus ‘m_program_ptr->size()’
    for (unsigned& i : m_vector)
    {
      ++i;

      // Need to carry over to next index?
      if (m_program_ptr->size() == i)
      {
        i = 0;
      }
      else
      {
        is_end = false;

        break;
      }
    }

    if (is_end)
      m_program_ptr = nullptr;

    return p; // good compilers will use RVO
  }
};
```

Listing 3.2: A nonstandard single-pass input iterator for finite partial strings

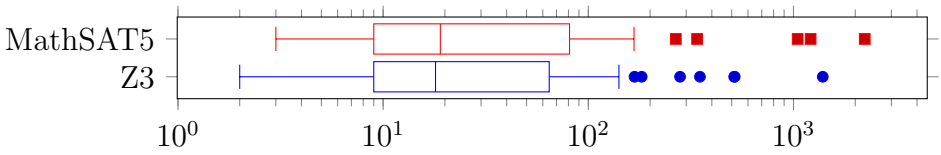


Figure 3.7: Log-scaled box plot of benchmark run-time (s).

the $\langle P \rangle$ grammar acts as a sufficient (but not necessary) condition for constructing elementary programs according to Definition 3.3.9. For example, the program

```
(Y + ((U , Z)| (V | (W | (X + (Y , X))))))
```

is an instance of the $\langle P \rangle$ grammar, comprising 15 syntax symbols.

As the first step in our experiments, we used Seed [HN11] to randomly generate approximately 2,000 pairs of elementary programs `P` and `Q`, for which we check whether `lfp<',>(P)` refines `lfp<',>(Q)`, i.e. `lfp<',>(P)<= lfp<',>(Q)` as illustrated by Example 3.4.2. This initial benchmark completed in less than 24 hours on a machine with a 2 GHz Intel Core i7 processor and 8 GB 1600 MHz DDR3 of main memory.

For further analysis, we selected from this initial benchmark 51 pairs that ran for more than 3 s and required at least one SMT solver call. For those 51 pairs, the number of iterations of the loop in Algorithm 3 is summarized by the histogram in Figure 3.6. On average, the loop iterates 2.4×10^6 times. This data point quantifies the large number of CHECK_EPR$(\mathcal{X}, \mathcal{K})$ calls, each of corresponds to solving an instance of an `QF_UFLIA` problem. Figure 3.6 therefore justifies our design decision to directly interact with the SMT solver through a programmatic interface, rather than the more standard but slower SMT-LIB format, which imposes additional overhead due to file I/O and text parsing. In fact, at the extremes, we see three instances in Figure 3.6 with over 10^7 loop iterations, another witness of the exponential number of partial strings that need to be considered by the decision procedure in the worst-case.

To better understand the impact on the underlying decision procedure, Figure 3.7 compares the run-time of each elementary program refinement check using MathSAT5 and Z3. This shows that the SMT solver can make one order of magnitude difference in run-time. Averaging the plotted run-time gives a mean of 133 s for MathSAT5 and 90 s for Z3 with a standard deviation of 373 s and 218 s, respectively.

Our experiments also quantify the benefit of our `LazyProgram` data structure that uses a vector of integers to represent partial strings in a program that is iteratively composed with itself. We experimentally confirmed that `LazyProgram` reduces the memory footprint two orders of magnitude, e.g. 25 MB versus 5 GB for even a single refinement check.

In summary, it is only through our lazy program optimization that we can comfortably run all 2,000 benchmarks within the available memory limits. In addition, our benchmarks quantify the large number of loop iterations even on relatively small programs (Figure 3.6), justifying the need for heuristics when higher performance is required. Lastly, the comparison of two SMT solvers shows that the efficiency of the underlying standard decision procedure (`QF_UFLIA`) can make up to one order of magnitude difference in run-time.

The significance of these experimental results derives from the fact that they identify two seemingly separate engineering concerns in a problem that has been unexplored so far: the design of an SMT solver that can efficiently deal with the partial string refinement problem through the use of uninterpreted functions and partial-order constraints, and a space efficient representation of sets of partial strings. The former was shown to make a one order of magnitude difference in time, whereas the latter reduced the space consumption by at least one order of magnitude. Notably, it is possible to to combine both improvements since they are implemented as separate standalone modules.

3.5 Concluding remarks

We have presented a least fixed point reduction theorem for a fragment of the partial string model of CKA. This result is significant because it justifies investigations into the existence of a normal form for the pomset language closed under least fixed point, sequential and concurrent composition operators as well as the exchange law.

Throughout this chapter, we emphasized those partial string constructions that helped us devise a decision procedure for our fragment of CKA that can leverage SAT/SMT solvers. Our experiments quantified two sources for the complexity of the EPR$^{\bowtie}$ decision problem, namely the partial string refinement check using an SMT solver and the exponential number of these checks even in the case of a small random data set. We showed how to compensate for this exponential explosion by implementing a lazy partial string iterator.

3.6 Bibliographic notes

The significance of partial-orders for the modelling of concurrency was early on recognized and has extensively flourished ever since in the theoretical computer science literature on this topic, e.g. [Pet66, Lam78, NPW81, Gra81, Pra86, Gis88].

The closest work to ours is Gischer's pomset model [Gis88] which is also generated from the downward-closure of sets of series-parallel pomsets. Such pomsets are more general than Mazurkiewicz traces [Maz87] because if a and b occur in parallel in a Mazurkiewicz trace, they cannot occur sequentially, whereas pomsets impose no such restriction [BK92]. The decidability of inclusion problems for pomset languages with star operators but without the exchange law has been most recently established [LS14]. More traditionally, recursion and pomsets were treated in the context of ultra-metric spaces [dBW90]. Winskel's event structures [Win82] are pomsets enriched with a conflict relation subject to certain conditions. Our partial-order abstraction of programs is firmly grounded on Ésik's recent work on infinite partial strings and their monotonic bijective morphisms [É02]. The fact that all these works use partial-orders to describe the dependency between events means that there is a close connection to the unfolding of Petri nets to occurrence nets, an active research area throughout the last four decades, e.g. [vGP09]. Lamport's two-arrow model can be derived by partitioning the events in a finite partial string x such that, for all sets of events A and B, $A \to B \triangleq \forall e \in A\colon \forall e' \in B\colon e \preceq_x e'$ and $A \dashrightarrow B \triangleq \exists e \in A\colon \exists e' \in B\colon e \preceq_x e'$ [Lam79b, Lam80]. These relations can be used to infer synchronization primitives from a proof of correctness [Lam97]. Throughout this chapter, our focus has been primarily on the ordering of events rather than checking the correctness of relaxed memory code with respect to an abstract implementation, cf. [BGMY12]. Recently, CKA has been extended with tests [Jip14] where extra conditions have to be imposed for the exchange law to hold.

Chapter 4

An asymptotically smaller partial-order encoding

Collaborators *This chapter is based on material that has been peer reviewed in [HK15, HHK15]. The latter of these two papers was written in conjunction with Liana Hadarean and Tim King.*

4.1 Introduction

For efficiency reasons, all modern computer architectures implement some form of weak memory semantics rather than sequential consistency [Lam79a]. As a result, a defining characteristic of weak memory architectures is that they violate interleaving semantics unless specific instructions are used to restore sequential consistency. In this chapter, we use partial strings to formally describe a form of relaxed sequential consistency (we call it *SC-relaxed consistency*) that follows the style of weak memory concurrency semantics that relax the program order of instructions per thread. An example of such a relaxation to the program order is TSO where a write followed by a read on the same memory location in the same thread can be reordered.

From a programmer's perspective, SC-relaxed consistency is about synchronizing threads by using release/acquire instructions. This synchronization mechanism is inspired by the release consistency model [GLL+90] and is an important building block for lock-free algorithms and low-latency, nonblocking asynchronous message passing. While release/acquire could be seen as generalizing the operations on a mutex, there are important differences: (i) unlike the lock and unlock operations on a mutex, a resource need not be released by the same thread that has acquired it, and (ii) release/acquire do not generally guarantee mutual exclusion. Instead, the

main purpose of release/acquire is to order accesses to the same memory location such that prior writes made to other memory locations by the thread executing the release become visible in the thread that performs the corresponding acquire action. The ordering induced between release and acquire on the same memory location is also known as the 'synchronizes-with' or 'global read-from' relation. This relation is determined by the execution of the program and can therefore vary between program runs. We show how to model this by further restricting the downward-closure of programs (Definition 3.3.1), providing the basis for SC-relaxed consistency.

We show that SC-relaxed consistency together with certain least upper bound constraints can be characterized by three extensively studied weak memory axioms, namely 'write coherence', 'global read-from' and 'from-read' [AMSS11, AKT13]. An important consequence of this characterization is an asymptotically smaller partial-order encoding that can be constructed in a quadratic (rather than a cubic) number of steps (Theorem 4.2.14). We show how to perform this construction in CBMC that implements the state-of-the-art cubic-size partial-order encoding [AKT13]. Our implementation provides the basis for an experimental comparison that offers new insights into the trade-offs involved when designing a partial-order encoding. In particular, we experimentally show that our quadratic-size encoding tends to slow down the SAT solver but can nevertheless significantly reduce the total run-time of CBMC on sufficiently large benchmarks, such as the OpenCores Ethernet MAC.

Contributions The main contributions of this chapter are threefold:

1. We define SC-relaxed consistency in terms of certain downward-closed sets of partial strings (Definition 4.2.3), and show that our interpretation corresponds to three fundamental weak memory axioms applied to a subset of reads and writes (Theorem 4.2.13), namely our notion of release/acquire events. This is an important correspondence because all three axioms underpin extensive experimental research into weak memory architectures [AMSS11].

2. We then prove that SC-relaxed consistency yields an asymptotically smaller quantifier-free first-order logic formula that can be constructed in only $O(N^2)$ steps (Theorem 4.2.14) compared to the state-of-the-art partial-order encoding for bounded model checking [AKT13] that requires $O(N^3)$ steps where N is the maximal number of reads and writes on the same shared memory location. This is significant because N can be prohibitively large when concurrent programs frequently share data. Another significance of our partial-order encoding is

that it provides the basis for new experiments in which the 'from-read' axiom is replaced by a new axiom.

3. To perform these experiments, we give a reference implementation in CBMC. By comparing the cubic-size and quadratic-size partial-order encoding using the same implementation, our experimental results provide new clues about how the 'from-read' axiom affects the performance of the concurrency analysis. In particular, we show that the new quadratic-size partial-order encoding drastically speeds up the construction of the propositional formula in CBMC such that nearly 99% of the total run-time is now used for SAT solving. We show that this can reduce the total run-time for sufficiently large problems, such as the OpenCores Ethernet MAC benchmark, where at most 30–40% of the total run-time was previously used for SAT solving.

Organization We first characterize a particular form of relaxed sequential consistency in terms of three weak memory axioms by Alglave et al. (Section 4.2). We then use this characterization to construct an asymptotically smaller partial-order encoding (Section 4.3). Finally, we show how to implement our quadratic-size partial-order encoding in CBMC that we experimentally compare to the cubic-size state-of-the-art partial-order encoding implemented in the same tool (Section 4.4).

4.2 Quadratic-size partial-order encoding

In this section, we explain how to asymptotically reduce the number of steps that are needed to construct a partial-order encoding of SC-relaxed consistency. Our formalism is the basis for the quadratic-size partial-order encoding in Section 4.3.

4.2.1 Memory accesses

We start by defining an alphabet that separates memory accesses into synchronizing and non-synchronizing ones. A synchronized store is called a *release*, whereas a synchronized load is called an *acquire*. The intuition behind release/acquire is that prior writes made to other memory locations by the thread executing the release become visible in the thread that performs the corresponding acquire action.

Definition 4.2.1 (Memory access alphabet)**.** Define $\langle LOAD \rangle \triangleq \{\mathsf{none}, \mathsf{acquire}\}$, $\langle STORE \rangle \triangleq \{\mathsf{none}, \mathsf{release}\}$ and $\langle BIT \rangle \triangleq \{0, 1\}$. Let $\langle ADDRESS \rangle$ and $\langle REG \rangle$ be

Thread $\mathtt{T}_1$		Thread $\mathtt{T}_2$
$v_0 := [b]_{\mathsf{acquire}}$	$\Vert$	$[a]_{\mathsf{none}} := 1$
$v_1 := [a]_{\mathsf{none}}$	$\Vert$	$[b]_{\mathsf{release}} := 1$

Figure 4.1: A concurrent system $\mathtt{T}_1 \parallel \mathtt{T}_2$ where the accesses on memory location b are synchronized, whereas those on memory location a are not.

disjoint sets of **memory locations** and **registers**, respectively. Let $load\_tag$ and $store\_tag$ be elements in $\langle LOAD \rangle$ and $\langle STORE \rangle$, respectively. Define the set of **load** and **store** labels, respectively, as follows:

$$\Gamma_{\mathsf{load},\, load\_tag} \triangleq \{load\_tag\} \times \langle REG \rangle \times \langle ADDRESS \rangle$$
$$\Gamma_{\mathsf{store},\, store\_tag} \triangleq \{store\_tag\} \times \langle ADDRESS \rangle \times \langle BIT \rangle.$$

Define the **memory access alphabet**, written Γ, to be the union of $\Gamma_{\mathsf{load,none}}$, $\Gamma_{\mathsf{load,acquire}}$, $\Gamma_{\mathsf{store,none}}$ and $\Gamma_{\mathsf{store,release}}$. Given $r \in \langle REG \rangle$, $a \in \langle ADDRESS \rangle$ and $b \in \langle BIT \rangle$, we write '$r := [a]_{load\_tag}$' for the label $\langle load\_tag, r, a \rangle$ in $\Gamma_{\mathsf{load},\, load\_tag}$; similarly, '$[a]_{store\_tag} := b$' is shorthand for the label $\langle store\_tag, a, b \rangle$ in $\Gamma_{\mathsf{store},\, store\_tag}$.

Let x be a partial string and e be an event in E_x. Then e is called a **load** or **store** whenever its label, $\alpha_x(e)$, is either in $\Gamma_{\mathsf{load},\, load\_tag}$ or $\Gamma_{\mathsf{store},\, store\_tag}$, respectively. A load or store event e is called a **non-synchronizing memory access** whenever $\alpha_x(e) \in \Gamma_{\mathsf{none}} \triangleq \Gamma_{\mathsf{load,none}} \cup \Gamma_{\mathsf{store,none}}$; otherwise, it is a **synchronizing memory access**. Let $a \in \langle ADDRESS \rangle$ be a memory location. An **acquire on** a is an event e such that $\alpha_x(e) = `r := [a]_{\mathsf{acquire}}'$ for some $r \in \langle REG \rangle$. Similarly, a **release on** $a$ is an event $e$ labelled by $`[a]_{\mathsf{release}} := b'$ for some $b \in \langle BIT \rangle$. A **release** and **acquire** is a release and acquire on some memory location, respectively.

We remark that, by disjointness of labels, loads and stores are always distinct. Also note that we limit our formalism to loads and stores of single bits because we are mainly interested in the partial ordering constraints for now. It is not difficult to generalize this to other data types. In fact, our implementation in Section 4.4 can handle all data types and arithmetic operators that are currently supported by CBMC [AKT13].

Example 4.2.2. Figure 4.1 shows the syntax of a program that consists of two threads $\mathtt{T}_1$ and $\mathtt{T}_2$. If we ignore the initialization of the memory $[a]$ and $[b]$ for now, this concurrent system can be directly modelled by the partial string p shown in Figure 3.1 where the labelling function, α_p, satisfies the following: $\alpha_p(e_0) = `v_0 := [b]_{\mathsf{acquire}}'$, $\alpha_p(e_1) = `v_1 := [a]_{\mathsf{none}}'$, $\alpha_p(e_2) = `[a]_{\mathsf{none}} := 1'$ and $\alpha_p(e_3) = `[b]_{\mathsf{release}} := 1'$. The labels

for the events e_0 and e_3 denote the fact that memory location b is accessed through acquire and release, respectively, whereas the events e_1 and e_2 access memory location a through non-synchronizing loads and stores. In the next subsection, we shall see that this leads to a data race. The notion of a data race will be formalized in Definition 4.2.3 below.

4.2.2 SC-relaxed programs

Given the memory access alphabet (Definition 4.2.1) from the previous subsection, we aim to refine our earlier conservative over-approximation of the happens-before relations (Definition 3.3.1) to get a particular form of release/acquire semantics. Towards this end, we restrict the downward closure of programs $\mathcal{X}$ in $\mathbb{P}$, in the sense of Definition 3.3.1, by requiring all partial strings in $\mathcal{X}$ to satisfy the following partial ordering constraints:

Definition 4.2.3 (SC-relaxed program)**.** A program $\mathcal{X}$ is called **SC-relaxed** if, for all $a \in \langle ADDRESS \rangle$ and partial string x in $\mathcal{X}$, the set of release events on a is totally ordered by $\preceq_x$ and, for every acquire $l \in E_x$ and release $s \in E_x$ on a, $l \preceq_x s$ or $s \preceq_x l$. If a program $\mathcal{X}$ contains a partial string x such that two events e and e' in x access the same memory location, at least one of which is a store, but neither $e \preceq_x e'$ nor $e' \preceq_x e$, then $\mathcal{X}$ has a **data race**. An SC-relaxed program is said to be **well-defined** if it contains no data races.

Henceforth, we denote loads and stores by l, l' and s, s', respectively. If s and s' are release events that modify the same memory location in an SC-relaxed program, the total ordering of these stores means that either s happens-before s', or vice versa. Furthermore, if l is an acquire and s is a release on the same memory location, either l happens-before s, or vice versa. SC-relaxed programs constitute a particular form of weak memory semantics because two acquire events l and l' on the same memory location may still happen concurrently in the sense that neither l happens-before l' nor l' happens-before l, in the same way non-synchronizing memory accesses are generally unordered. The absence of these orderings can introduce data races, as explained next.

A *data race* means that two threads access the same memory location, at least one of which modifies the memory location, but neither memory access is ordered through the happens-before relation. For example, $[a]_{\mathsf{none}} := 1 \parallel [a]_{\mathsf{none}} := 0$ has a data race because both writes are non-synchronizing and therefore unordered by the happens-before relation, i.e. they happen concurrently. We emphasize that our notion

of data races is well-established (e.g. [AHMN91, FF09, BCM10, DWS+12]) and should not be confused with the same terminology in the literature on sequential consistency where a data race refers to all conflicting memory accesses. By contrast, we only consider conflicting memory accesses to be a data race when they are unordered by the happens-before relation. This can only happen if the memory accesses are non-synchronized. In optimizing compilers that target weak memory architectures, data races in this sense can generally cause undefined program behaviour [Adv10, AB10, BA12]. In contrast to traditional interleaving semantics, we therefore only consider an SC-relaxed program to be well-defined if it contains no data races. Since our focus is not on data race detection, we assume that the SC-relaxed program is well-defined, i.e. it is data race free. A sufficient condition for data race freedom is to only use release/acquire instructions.

Example 4.2.4. The SC-relaxed program of the set $\mathcal{X} = \{x\}$ where x is the partial string described in Example 4.2.2 illustrates the SC-relaxed semantics of the concurrent system in Figure 4.1. In particular, the SC-relaxed program contains the N-shaped partial string in Figure 3.4. In fact, this N-shaped partial string corresponds to a data race in $\mathtt{T}_1 \parallel \mathtt{T}_2$ because the non-synchronizing memory accesses on memory location a happen concurrently. Another way of explaining this data race is by considering the following permitted linearization of the N-shaped partial string:

$$v_0 := [b]_{\mathsf{acquire}}; [a]_{\mathsf{none}} := 1; v_1 := [a]_{\mathsf{none}}; [b]_{\mathsf{release}} := 1$$

where both memory accesses on location a are unordered through the happens-before relation because there is no release instruction separating $[a]_{\mathsf{none}} := 1$ from $v_1 := [a]_{\mathsf{none}}$. One way of fixing this data race is by changing the last instruction in thread $\mathtt{T}_1$ to the following: **if** $v_0 = 1$ **then** $v_1 := [a]_{\mathsf{none}}$. The resulting concurrent system is shown in Figure 4.8. Since CKA supports non-deterministic choice with the $\cup$ binary operator (recall Theorem 3.3.6), it is not difficult to give semantics to such concurrent systems, particularly if we introduce 'assume' labels into the alphabet in Definition 4.2.1, similar to [WKGG09]. All this will be revisited in Example 4.3.4 where we show how to symbolically encode the control-flow statement in Figure 4.8.

4.2.3 Weak memory axioms

We ultimately want to show that the conjunction of three existing weak memory axioms studied as part of [AMSS12] fully characterizes our particular interpretation of relaxed sequential consistency, thereby paving the way for Theorem 4.2.14. For

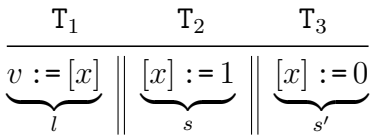


Figure 4.2: Concurrent system where three threads access the same memory location.

this, we recall the following memory axioms which can be thought of as relations on loads and stores on the same memory location. Here, however, we simplify the memory axioms such that they are not defined as separate relations but as part of the same partial-order, the so-called *happens-before relation*. While this simplification prevents us from expressing the full range of weak memory models that Alglave et al. have studied, our formalism retains essential characteristics of the problem, including the requirement that each memory access per location is sequentially consistent.[1] This frees us to focus on the algorithmic aspects of partial-order encodings rather than hardware-specific or language-specific semantics questions.

Definition 4.2.5 (Weak memory axioms)**.** Let x be a finite partial string. The **read-from** axiom asserts the existence of a **read-from function**, $\mathsf{rf}\colon E_x \to E_x$, that maps every load to a store on the same memory location. The **synchronizes-with** axiom holds whenever, for every load l and store s, $\mathsf{rf}(l) = s$ implies $s \preceq_x l$. The **write coherence** axiom holds if all stores s, s' on the same memory location are totally ordered by $\preceq_x$. The **from-read** axiom holds whenever, for all loads l and stores s, s' on the same memory location, if $\mathsf{rf}(l) = s$ and $s \prec_x s'$, then $l \preceq_x s'$. When convenient, we restrict these axioms to a subset of load and store events. For example, if we say that the 'synchronizes-with' axiom holds *with respect to all release/acquire events*, we mean that, for all acquire l and release s events, $\mathsf{rf}(l) = s$ implies $s \preceq_x l$.

The weak memory axioms in Definition 4.2.5 capture essential characteristics of the partial-order encodings we study. We can illustrate these weak memory axioms through the more commonly used diagrams in [AMSS12]. For this purpose, consider the concurrent system in Figure 4.2. By the 'read-from' axiom, we can assume that, for every memory location, there is a store that initializes it. The read-from function determines then the store from which a load gets its value. For example, ignoring initialization events, there are two possible 'read-from' functions for the concurrent system in Figure 4.2, illustrated by the two diagrams in Figure 4.3. In particular, Figure 4.4a and 4.4b show the case where l gets its value from s and s', respectively.

[1] Alglave et al. formalize the sequential consistency guarantee per memory location by requiring the union of four relations they define, namely 'po-loc', 'co', 'rf' and 'fr', to be acyclic.

Figure 4.3: Two ‘read-from’ functions for the concurrent system in Figure 4.2.

Suppose we select the ‘read-from’ function in Figure 4.4a. This means that the load l in $\mathtt{T}_1$ reads-from the store s in $\mathtt{T}_2$, implying that the final register value is $v = 1$.

Remark 4.2.6. *We remark that in earlier work [AMSS12] ‘read-from’ is defined to be a relation for conformity reasons. Assuming that ‘synchronizes-with’, ‘write coherence’ and ‘from-read’ hold for all events, it is immediate by antisymmetry that ‘read-from’ is a function, which we have made explicit in Definition 4.2.5.*

The ‘write coherence’ axiom, in turn, ensures that all stores on the same memory location are totally ordered. This axiom captures computer architectures in which “all writes to the same location are serialized in some order and are performed in that order with respect to any processor” [GLL+90]. Since we define all axioms, including ‘write coherence’, in terms of a single happens-before relation, ‘write coherence’ is different from the modification order (‘mo’) on atomics in C++14 [ISO14], and the ‘ws’ or ‘co’ relations in [AMSS12, AMT14] because neither ‘mo’, ‘ws’ nor ‘co’ is generally a subset of a single happens-before relation. The rationale for ‘mo’, ‘ws’ and ‘co’ to be not included in the happens-before relation is that it would be too restrictive for certain kinds of weak memory semantics, particularly memory fences. This can be illustrated by the archetypal `2+2w` litmus test [AMT14] in which a separate order constraint governs how ‘co’ interacts with fences. However, since we concentrate on algorithmic aspects of partial-order encodings, we deem the semantical restrictions that our formalism imposes to be acceptable. As an example of ‘write coherence’, in the case of the two stores s and s' in Figure 4.2, either s happens-before s', or vice versa. If we assume that ‘co’ forms part of a single happens-before relation, we can illustrate both possibilities through the two diagrams in Figure 4.4.

Figure 4.4: For the concurrent system in Figure 4.2, s happens-before s', or vice versa.

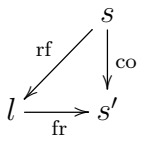


Figure 4.5: The 'from-read' axiom on the same memory location where store s happens-before another store s', and load l reads-from s.

The 'synchronizes-with' axiom says that if a load reads-from a store (necessarily on the same memory location), then the store happens-before the load. This is also known as the 'global read-from' axiom [AMSS12]. For the 'read-from' function in Figure 4.4a, we get that $s \preceq_x l$ by the 'synchronizes-with' axiom.

Finally, the 'from-read' axiom is derived from both the 'write coherence' and 'read-from' axiom. This way, the 'from-read' axiom makes explicit how a value in memory is overwritten. For this reason, it is also known as the 'conflict relation' [BDM13]; it should not be confused though with Winskel's conflict relation [NPW81] which could be used for this purpose [JR16] but also for modelling non-deterministic control flow, as mentioned in Section 3.3. In our running example, since l reads-from s, and assuming s happens-before s', the 'from-read' axiom says that l happens-before s', as depicted by the diagram in Figure 4.5. Informally, this diagram means that we get the following sequential program:

$$\underbrace{[x] := 1}_{s}; \underbrace{v := [x]}_{l}; \underbrace{[x] := 0}_{s'}.$$

That is to say, the second s' overwrites the value read by the proceeding load l. Note that it is, in fact, the 'from-read' axiom that causes the cubic size complexity in the state-of-the-art partial-order encoding [AKT13]: the number of steps performed by CBMC to construct the partial-order encoding is proportional in the number of iterations of three nested loops, one for each of the three universal quantifiers in the 'from-read' axiom. The purpose of the remaining part of this section is to show how to reduce this complexity to be merely quadratic.

4.2.4 SC-relaxed consistency

We next characterize the three weak memory axioms from the previous subsection in terms of certain least upper bound constraints and SC-relaxed programs. We start by deriving from the three weak memory axioms the notion of SC-relaxed programs.

Proposition 4.2.7 (Sufficient SC-relaxations)**.** *For all programs $\mathcal{X}$ in $\mathbb{P}$, if, for each finite partial string x in $\mathcal{X}$, the 'synchronizes-with', 'write coherence' and 'from-read' axioms hold with respect to all release and acquire events in E_x on the same memory location, then $\mathcal{X}$ is an SC-relaxed program.*

Proof. Let $a \in \langle ADDRESS \rangle$ be a memory location, l be an acquire on a and s' be a release on a. By 'write coherence' on release/acquire events, it remains to show $l \preceq_x s'$ or $s' \preceq_x l$. Since the 'read-from' function is total, $\mathsf{rf}(l) = s$ for some release s on a. By the 'synchronizes-with' axiom, $s \preceq_x l$. We therefore assume $s \neq s'$. By 'write coherence', $s \prec_x s'$ or $s' \prec_x s$. The former implies $l \preceq_x s'$ by the 'from-read' axiom, whereas the latter implies $s' \preceq_x l$ by transitivity. This proves, by case analysis, that $\mathcal{X}$ is an SC-relaxed program. □

We need to prove some form of converse of the previous implication in order to characterize SC-relaxed semantics in terms of the three weak memory axioms in Definition 4.2.5. For this purpose, we define the following least upper bound:

Definition 4.2.8 (SC-relaxed consistency)**.** Let $a \in \langle ADDRESS \rangle$ be a memory location and x be a finite partial string in P_f. For all loads $l \in E_x$ on a, define the following set of store events:

$$\mathcal{H}_x(l) \triangleq \{s \in E_x \mid s \preceq_x l \text{ and } s \text{ is a store on memory location } a\}.$$

The 'read-from' function rf is said to satisfy **weak read consistency** if, for all loads $l \in E_x$ and stores $s \in E_x$ on memory location a, the least upper bound $\bigvee \mathcal{H}_x(l)$ exists, and $\mathsf{rf}(l) = s$ implies $\bigvee \mathcal{H}_x(l) \preceq_x s$; furthermore, **strong read consistency** implies $\mathsf{rf}(l) = s = \bigvee \mathcal{H}_x(l)$. We speak of *SC-relaxed consistency* whenever a program is SC-relaxed and it satisfies strong read consistency.

By the next proposition, a natural sufficient condition for the existence of the least upper bound $\bigvee \mathcal{H}_x(l)$ is the finiteness of the partial strings in P_f and the total ordering of all stores on the same memory location from which the load l reads, i.e. 'write coherence'. This could be generalized to well-ordered sets.

Proposition 4.2.9 (Weak read consistency existence)**.** *For all finite partial strings x in P_f, 'write coherence' on memory location a implies that $\bigvee \mathcal{H}_x(l)$ exists for all loads l on memory location a.*

We remark that $\bigvee \mathcal{H}_x(l) = \bot$ if $\mathcal{H}_x(l) = \emptyset$; alternatively, to avoid that $\mathcal{H}_x(l)$ is empty, we implicitly assume that programs are constructed such that their partial

strings have minimal store events that initialize all memory locations. This assumption can be satisfied by rewriting a program $\mathcal{P}$ to $\mathcal{Z};\mathcal{P}$ where $\mathcal{Z} = \{z\}$ consists of a single partial string z whose events initialize all subsequently used memory locations to some predefined value, typically zero.

Proposition 4.2.10 (Weak read consistency characterization)**.** *Write coherence implies that weak read consistency is equivalent to the following: for all loads l and stores s, s' on memory location $a \in \langle ADDRESS \rangle$, if $\mathsf{rf}(l) = s$ and $s' \preceq_x l$, then $s' \preceq_x s$.*

Proof. By 'write coherence', $\bigvee \mathcal{H}_x(l)$ exists, and $s' \preceq_x \bigvee \mathcal{H}_x(l)$ because $s' \in \mathcal{H}_x(l)$ by assumption $s' \preceq_x l$ and Definition 4.2.8. By assumption of weak read consistency, $\bigvee \mathcal{H}_x(l) \preceq_x s$. From transitivity follows $s' \preceq_x s$.

Conversely, assume $\mathsf{rf}(l) = s$. Let s' be a store on a such that $s' \in \mathcal{H}_x(l)$. Thus, by hypothesis, $s' \preceq_x s$. Since s' is arbitrary, s is an upper bound. Since the least upper bound is well-defined by 'write coherence', $\bigvee \mathcal{H}_x(l) \preceq_x s$. □

Weak read consistency therefore says that if a load l reads-from a store s and another store s' on the same memory location happens-before l, then s' happens-before s. This implies the next proposition.

Proposition 4.2.11 (From-read characterization)**.** *For all SC-relaxed programs in $\mathbb{P}$, weak read consistency with respect to release/acquire events is equivalent to the 'from-read' axiom with respect to release/acquire events.*

We can characterize strong read consistency as follows:

Proposition 4.2.12 (Strong read consistency characterization)**.** *Strong read consistency is equivalent to weak read consistency and the 'synchronizes-with' axiom.*

Proof. Let x be a partial string in P_f. Let l be a load and s be a store on the same memory location. The forward implication is immediate from $\bigvee \mathcal{H}_x(l) \preceq_x l$.

Conversely, assume $\mathsf{rf}(l) = s$. By 'synchronizes-with', $s \preceq_x l$, whence $s \in \mathcal{H}_x(l)$. By definition of least upper bound, $s \preceq_x \bigvee \mathcal{H}_x(l)$. Since $s \succeq_x \bigvee \mathcal{H}_x(l)$, by hypothesis, and $\preceq_x$ is antisymmetric, we conclude $s = \bigvee \mathcal{H}_x(l)$. □

Theorem 4.2.13 (SC-relaxed consistency characterization)**.** *For every program $\mathcal{X}$ in $\mathbb{P}$, $\mathcal{X}$ is SC-relaxed where, for all finite partial strings x in $\mathcal{X}$ and acquire events l in E_x, $\mathsf{rf}(l) = \bigvee \mathcal{H}_x(l)$, if and only if the 'synchronizes-with', 'write coherence' and 'from-read' axioms hold for all finite partial strings x in $\mathcal{X}$ with respect to all release/acquire events in E_x on the same memory location.*

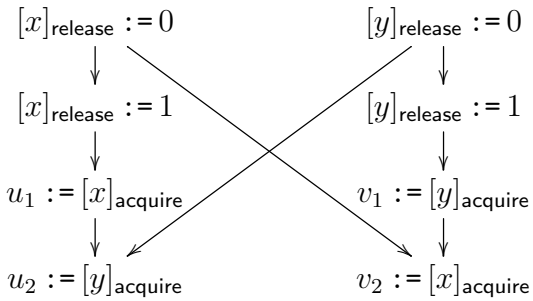
Thread $\mathtt{T}_1$	Thread $\mathtt{T}_2$	Thread $\mathtt{T}_3$	Thread $\mathtt{T}_4$
$[x]_{\mathsf{release}} := 1$	$[y]_{\mathsf{release}} := 1$	$u_1 := [x]_{\mathsf{acquire}}$ $u_2 := [y]_{\mathsf{acquire}}$	$v_1 := [y]_{\mathsf{acquire}}$ $v_2 := [x]_{\mathsf{acquire}}$

Figure 4.6: Initially, $[x]_{\mathsf{release}} := 0$ and $[y]_{\mathsf{release}} := 0$. SC-relaxed consistency forbids that $u_1 = v_1 = 1$ and $u_2 = v_2 = 0$.

Proof. Assume $\mathcal{X}$ is an SC-relaxed program according to Definition 4.2.3. Let x be a partial string in $\mathcal{X}$ and l be an acquire in the set of events E_x. By Proposition 4.2.9, $\bigvee \mathcal{H}_x(l)$ exists. Assume $\mathsf{rf}(l) = \bigvee \mathcal{H}_x(l)$. Since l is arbitrary, this is equivalent to assuming strong read consistency. Since release events are totally ordered in $\preceq_x$, by assumption, it remains to show that the 'synchronizes-with' and 'from-read' axioms hold. This follows from Proposition 4.2.12 and 4.2.11, respectively.

Conversely, assume the three weak memory axioms hold on x with respect to all release/acquire events in E_x on the same memory location. By Proposition 4.2.7, $\mathcal{X}$ is an SC-relaxed program. Therefore, by Proposition 4.2.11 and 4.2.12, $\mathsf{rf}(l) = \bigvee \mathcal{H}_x(l)$, proving the equivalence. □

We emphasize that SC-relaxed consistency cannot express all forms of weak memory. For example, SC-relaxed consistency forbids $u_1 = v_1 = 1$ and $u_2 = v_2 = 0$ given the concurrent system in Figure 4.6 even though it could be allowed on some weaker architectures such as ARM. To see what causes the register values to be forbidden, consider the following partial string:



By strong read consistency (Definition 4.2.8), the registers satisfy $u_1 = v_1 = 1$ and $u_2 = v_2 = 0$. However, it is impossible to construct an SC-relaxed program according to Definition 4.2.3 because we cannot order the events that are labelled by $[x]_{\mathsf{release}} := 1$ and $v_2 := [x]_{\mathsf{acquire}}$ as well as $[y]_{\mathsf{release}} := 1$ and $v_1 := [y]_{\mathsf{acquire}}$ without either (i) violating antisymmetry, (ii) changing the register values, or (iii) having to relax the partial ordering in some way.

While the state-of-the-art weak memory encoding is cubic in size [AKT13], the

Algorithm 4 Cubic encoding

for $r \in \mathsf{R}_x$ **do**
  for $w \in \mathsf{W}_x$ **do**
    for $w' \in \mathsf{W}_x$ **do**
      $\textsc{Encode}^3(r,\, w,\, w')$

Algorithm 5 Quadratic encoding

for $r \in \mathsf{R}_x$ **do**
  for $w \in \mathsf{W}_x$ **do**
    $\textsc{Encode}^2(r,\, w,\, s_r)$

Figure 4.7: The crux behind our asymptotically faster algorithm for constructing a partial-order encoding of SC-relaxed consistency.

previous theorem has as immediate consequence that there exists an asymptotically smaller weak memory encoding with only a quadratic number of partial-order constraints that need to be explicitly constructed.

Theorem 4.2.14 (Quadratic-size weak memory encoding)**.** *There exists a quantifier-free first-order logic formula that has a quadratic number of partial-order constraints and is equisatisfiable to the cubic-size encoding [AKT13] of the conjunction of the 'write coherence', 'from-read' and 'global read-from' axioms.*

Proof sketch. Rather than instantiating the three universally quantified events in the 'from-read' axiom, we symbolically encode the least upper bound of weak read consistency instead. This is illustrated by the pseudo-code in Figure 4.7 where the cubic-size encoding (Algorithm 4) uses a triply nested loop over release/acquire events on memory location x, whereas the quadratic-size encoding (Algorithm 5) requires only a doubly nested loop over release/acquire events on memory location x. This asymptotically reduces the number of steps required for constructing the partial-order encoding of SC-relaxed consistency. This asymptotic reduction is achieved by introducing a new symbolic variable, written sup_r, for every acquire event r. The details of $\textsc{Encode}^2(r,\, w,\, \mathsf{sup}_r)$ are summarized by Figure 4.9 in Section 4.3 whose correctness follows from Theorem 4.2.13. □

In short, the asymptotic reduction in the number of partial-order constraints is due to a new symbolic encoding for how values are being overwritten in memory: the current cubic-size formula [AKT13] encodes the 'from-read' axiom (Definition 4.2.5), whereas the proposed quadratic-size formula encodes a certain least upper bound (Definition 4.2.8). We emphasize that this formulation is in terms of release/acquire events rather than machine-specific accesses as in [AKT13]. The construction of the quadratic-size encoding, therefore, is generally only applicable if we can translate the machine-specific reads and writes in a shared memory program to acquire and release

events, respectively. This may require the program to be data race free, as illustrated in Example 4.2.4. We emphasize that data race freedom can be trivially satisfied if each instruction in the program under scrutiny is assumed to be synchronized.

Furthermore, as mentioned in the introduction of this chapter, the primary application of Theorem 4.2.14 is in the context of bounded model checking. Recall that bounded model checking assumes that all loops in the shared memory program under scrutiny have been unrolled (the same restriction as in [AKT13]). This makes it possible to symbolically encode branch conditions, thereby alleviating the need to explicitly enumerate each finite partial string in an elementary program. The details of this encoding are shown next.

4.3 Symbolic encoding of SC-relaxed consistency

Let `P` be a C program that can be compiled into CPROVER's 'GOTO' intermediate representation.[2] In this section, we explain how to construct the cubic-size and quadratic-size partial-order encoding of `P` provided it satisfies the following conditions:

1. `P`'s weak memory concurrency semantics equates to SC-relaxed consistency;
2. every shared memory location accessed by `P` is known at compile-time;
3. `P` is well-structured and all recursive procedures as well as loops are bounded.

Recall that the intuition behind (well-defined) SC-relaxed consistency is that (i) all synchronized writes on the same shared memory location are totally ordered in the happens-before relation, (ii) all synchronized reads r and writes w on the same shared memory location satisfy that r happens-before w, or vice versa, and (iii) there are no data races in the sense that all (non-synchronized) conflicting memory accesses are ordered through the happens-before relation. Our second assumption guarantees that all reads and writes in `P` can be unambiguously associated with a concrete memory location. This restriction could be relaxed by adding pointer alias constraints to the partial-order encoding, which could in theory be formalized by the thread-modular summarization transformation rules in [SW10]. In practice, more sophisticated pointer logics may be needed, e.g. [Vaf09]. The final assumption means that every recursive procedure and loop needs to be only analyzed up to a bounded depth. In the sequential case, this makes it possible to fully verify `P` with a bounded

[2] http://www.cprover.org/svn/cbmc/releases/cbmc-5.2/src/goto-programs/goto_program_template.h

model checker. For concurrent programs, it furthermore implies that there is only a bounded number of threads that can be spawned by `P`.

Based on these assumptions, we could therefore think of `P` as an elementary program (Definition 3.3.9). However, it would be prohibitively expensive to reason about each finite partial string on an individual basis. We therefore aim to symbolically encode multiple partial strings instead. For this, we give two partial-order encodings, denoted by $\mathcal{E}^3$ and $\mathcal{E}^2$. The former mirrors a form of the cubic-size encoding from [AKT13], whereas the latter implements the quadratic-size encoding from Theorem 4.2.14. In both cases, the generated formula is satisfiable if and only if the safety property in the SC-relaxed program can be violated up to the loop unwinding bound. Another commonality of both $\mathcal{E}^3$ and $\mathcal{E}^2$ is that they can be divided into the following three parts:

1. *clock constraints* that partially order memory accesses;
2. *value constraints* that determine what values are read or written by the program if those clock constraints hold;
3. *selection constraints* that associate each read to a specific write event, thereby symbolically encoding the 'read-from' axiom.

We therefore parametrize both partial-order encodings by three logical theories: $\mathcal{T}_C$ for encoding the clock constraints, $\mathcal{T}_V$ for encoding constraints on the symbolic program values, and $\mathcal{T}_S$ for encoding selection constraints. While the formula generated by each encoding requires that these three theories be combined, we explain each of them in turn.

Intuitively, $\mathcal{T}_V$ is used to logically encode all arithmetic instructions that are used by the program `P` under scrutiny. This makes it possible to encode, what the authors in [AKT13] call, 'symbolic events'. For this, we assume that $\mathcal{T}_V$'s signature can encode a decidable fragment of common machine arithmetic such as Presburger or bit-vector arithmetic. In particular, depending on the required level of precision, $\mathcal{T}_V$ could be instantiated to, say, the first-order logic theory for linear integer arithmetic or fixed-size bit-vectors. Given $\mathcal{T}_V$, we represent the C program under scrutiny, `P`, by the following mathematical structure:

Definition 4.3.1 (Shared memory program structure)**.** A *shared memory program structure* is a tuple

$$P = \langle E, \ll, \alpha, val, guard \rangle$$

where E is a set of *events*, $\ll$ is a partial-order on E called the *preserved program order* (*PPO*), α is a function that labels events according to the memory location they access, $guard : E \to \mathcal{T}_V$-formulas and $val : E \to \mathcal{T}_V$-terms are two functions that map events to $\mathcal{T}_V$-formulas and $\mathcal{T}_V$-terms, respectively.

By the above assumptions, the set of events, E, is finite. Moreover, since we assume that all accessed memory locations are known at compile-time, the labelling function α is uniquely determined by the read and write instructions in $\mathtt{P}$. Therefore, the set of events E in a shared memory program structure P can be partitioned into reads R_x and writes W_x on memory location $x \in \langle ADDRESS \rangle$. From now on, we omit α when it is clear from the context what memory is accessed by an event. The intuition behind PPO is that it determines what events cannot be reordered in any execution of the program [AMSS12, AKT13]. For sequentially consistent programs, PPO corresponds to the order of instructions in each thread. Note that $\langle E, \ll \rangle$ may be relaxed for weaker forms of consistency such as TSO and PSO, e.g. [AKT13]. Intuitively, given an event e in E, $guard(e)$ denotes the necessary condition for e to be enabled. Finally, $val(e)$ denotes a symbolic value that is read or written by a particular instruction in $\mathtt{P}$ depending on whether e is a read or write event.

Example 4.3.2. To represent the concurrent system in Figure 4.1 as a shared memory program structure, we can choose the set of events, E, and their labels to be the same as for Example 4.2.2. For the PPO then, we can reuse the partial ordering in Figure 3.1. Since there are no control-flow statements, $guard(e)$ maps all events e in E to '**true**' in the theory $\mathcal{T}_V$. The val function maps the write events e_2 and e_3 to the numeric constant '1' in the theory $\mathcal{T}_V$.

The remaining two theories $\mathcal{T}_C$ and $\mathcal{T}_S$ concern the interaction of events in E. To express this, we assume that $\mathcal{T}_C$'s signature includes strict and non-strict partial-order relations, denoted by $\prec$ and $\preceq$, respectively, whereas $\mathcal{T}_S$ is a theory of uninterpreted functions and equality.

We denote by $\mathcal{T}$ the standard combined theory $\mathcal{T}_C + \mathcal{T}_V + \mathcal{T}_S$. We define the following constraint variables for every event in E:

Definition 4.3.3 (Constraint variables)**.** Given an event e in E, let c_e and s_e be a $\mathcal{T}_C$-variable (*clock variable*) and $\mathcal{T}_S$-variable (*selection variable*), respectively. For each read $r \in \mathsf{R}$, let rv_r be a $\mathcal{T}_V$-variable, called *read variable*.

Here is an example that shows how to use those constraint values to represent the concurrent system in Figure 4.8 which features a control-flow statement.

Thread $\mathtt{T}_1$		Thread $\mathtt{T}_2$
$v_0 := [b]_{\mathsf{acquire}}$	$\Vert$	$[a]_{\mathsf{none}} := 1$
if $v_0 = 1$ **then** $v_1 := [a]_{\mathsf{none}}$	$\Vert$	$[b]_{\mathsf{release}} := 1$

Figure 4.8: A concurrent system $\mathtt{T}_1 \parallel \mathtt{T}_2$ with a control-flow statement.

Example 4.3.4. Reconsider Example 4.2.4 where we suggested to fix the data race in Figure 4.1 by using a control-flow statement. The program in Figure 4.8 implements this suggestion. The set of events and their partial ordering is the same as for Example 4.3.2. However, what changes is the definition of the *guard* function that now satisfies $\mathit{guard}(e_1) = \text{`}\mathsf{rv}_{e_0} = 1\text{'}$. This means that the guard of event e_1 is the equality $\mathcal{T}_V$-term which says that the read value of event e_0 is equal to '1'. This equality corresponds to the control-flow condition that checks whether register v_0 is equal to '1'. For all other events e such that $e \neq e_1$, $\mathit{guard}(e) = \text{`}\mathbf{true}\text{'}$.

Equipped with a shared memory program structure (Definition 4.3.1) and constraint variables (Definition 4.3.3), Figure 4.9 shows how to generate the cubic-size $\mathcal{E}^3$ and quadratic-size $\mathcal{E}^2$ partial-order encoding for a given shared memory program structure $P = \langle E, \ll, \alpha, \mathit{val}, \mathit{guard}\rangle$. The first four formulas, **PPO**, $\mathbf{WW}[x]$, $\mathbf{RW}[x]$, and $\mathbf{RF_{TO}}[x]$, are shared by $\mathcal{E}^3$ and $\mathcal{E}^2$. The constraint **PPO** encodes the preserved program order $\ll$. The remaining constraints are with respect to some concrete memory location x that is inferred from the labelling function α. To model the information flow in the program, we encode the 'read-from' axiom which, by Definition 4.2.5, says that there exists a function from R_x to W_x for every memory location x. We model this function through the selection variables s_r and s_w, for each read $r \in \mathsf{R}_x$ and write $w \in \mathsf{W}_x$, together with the equality $\mathsf{s}_r = \mathsf{s}_w$. The $\mathbf{RF_{TO}}$ constraints ensures that at least one such equality holds for every read, thereby ensuring that the 'read-from' function is total. The intuition is that the value of a write event $w \in \mathsf{W}_x$ is observed by a read event $r \in \mathsf{R}_x$ if and only if $\mathsf{s}_r = \mathsf{s}_w$. By defining read variables, the function val can map a write event w in W to a $\mathcal{T}_V$-term $\mathit{val}(w)$. This is the basis of the $\mathbf{RF}^3$ and $\mathbf{RF}^2$ constraints that express the following: if $\mathsf{s}_w = \mathsf{s}_r$ holds in the theory $\mathcal{T}_S$, then the $\mathcal{T}_V$-variable rv_r is equal to the term $\mathit{val}(w)$. Informally, therefore, the equality $\mathsf{s}_w = \mathsf{s}_r$ says that a read event r is 'selected' so that the input value associated with r is equal to the output of a write event w. **WW** encodes that all writes on the same shared memory location are totally ordered in the happens-before relation and cannot have the same selection value, and **RW** encodes that every read r and write w on the same shared memory location satisfy that r happens-before w, or vice versa. Note that if $\prec$ is a total order, then **WW** is equivalent to the clock and selection

$$\mathbf{PPO} \triangleq \bigwedge \{(guard(e) \wedge guard(e')) \Rightarrow (\mathsf{c}_e \prec \mathsf{c}_{e'}) \mid e, e' \in E \colon e \ll e'\}$$

$$\mathbf{WW}[x] \triangleq \bigwedge \{(\mathsf{c}_w \prec \mathsf{c}_{w'} \vee \mathsf{c}_{w'} \prec \mathsf{c}_w) \wedge \mathsf{s}_w \neq \mathsf{s}_{w'} \mid w, w' \in \mathsf{W}_x \wedge w \neq w'\}$$

$$\mathbf{RW}[x] \triangleq \bigwedge \{\mathsf{c}_w \prec \mathsf{c}_r \vee \mathsf{c}_r \prec \mathsf{c}_w \mid w \in \mathsf{W}_x \wedge r \in \mathsf{R}_x\}$$

$$\mathbf{RF_{TO}}[x] \triangleq \bigwedge \{guard(r) \Rightarrow \bigvee \{\mathsf{s}_w = \mathsf{s}_r \mid w \in \mathsf{W}_x\} \mid r \in \mathsf{R}_x\}$$

$$\mathbf{RF}^3[x] \triangleq \bigwedge \{(\mathsf{s}_w = \mathsf{s}_r) \Rightarrow (guard(w) \wedge val(w) = \mathsf{rv}_r \wedge \mathsf{c}_w \prec \mathsf{c}_r) \mid r \in \mathsf{R}_x \wedge w \in \mathsf{W}_x\}$$

$$\mathbf{FR}[x] \triangleq \bigwedge \{(\mathsf{s}_w = \mathsf{s}_r \wedge \mathsf{c}_w \prec \mathsf{c}_{w'} \wedge guard(w')) \Rightarrow (\mathsf{c}_r \prec \mathsf{c}_{w'}) \mid w, w' \in \mathsf{W}_x \wedge r \in \mathsf{R}_x\}$$

$$\mathcal{E}^3 \triangleq \bigwedge \{\mathbf{RF_{TO}}[x] \wedge \mathbf{RF}^3[x] \wedge \mathbf{FR}[x] \wedge \mathbf{WW}[x] \wedge \mathbf{RW}[x] \mid x \in \langle ADDRESS \rangle\} \wedge \mathbf{PPO}$$

$$\mathbf{RF}^2[x] \triangleq \bigwedge \{(\mathsf{s}_w = \mathsf{s}_r) \Rightarrow (\mathsf{c}_w = \mathsf{sup}_r \wedge guard(w) \wedge val(w) = \mathsf{rv}_r \wedge \mathsf{c}_w \prec \mathsf{c}_r) \mid r \in \mathsf{R}_x \wedge w \in \mathsf{W}_x\}$$

$$\mathbf{SUP}[x] \triangleq \bigwedge \{(\mathsf{c}_w \preceq \mathsf{c}_r \wedge guard(w)) \Rightarrow (\mathsf{c}_w \preceq \mathsf{sup}_r) \mid r \in \mathsf{R}_x \wedge w \in \mathsf{W}_x\}$$

$$\mathcal{E}^2 \triangleq \bigwedge \{\mathbf{RF_{TO}}[x] \wedge \mathbf{RF}^2[x] \wedge \mathbf{SUP}[x] \wedge \mathbf{WW}[x] \wedge \mathbf{RW}[x] \mid x \in \langle ADDRESS \rangle\} \wedge \mathbf{PPO}$$

Figure 4.9: Given a shared memory program structure $P = \langle E, \ll, val, guard \rangle$, $\mathcal{E}^3$ and $\mathcal{E}^2$ encode P's SC-relaxed consistency with a cubic and quadratic number of constraints, respectively.

variables being distinct. (In practice, the s_w variables are optimized out as distinct constants.) Note that the same is not true for **RW**, however, because two reads can have the same values for their clock variables.

The main difference between $\mathcal{E}^3$ and $\mathcal{E}^2$ is how they encode values being overwritten in memory. A read r in R_x can read from a write w in W_x if w is the most recent write to x that happens-before r. In the case of $\mathcal{E}^3$, this is encoded by **FR** that corresponds to the 'from-read' axiom [AMSS12, AKT13]. This formula introduces a cubic number of constraints. By contrast, $\mathcal{E}^2$ encodes the **SUP** constraint that requires only a quadratic number of constraints. For this, **SUP** introduces a new variable sup_r for every read r in R_x to encode the least upper bound (supremum) of all writes in W_x that happen-before r. We therefore call sup_r a *supremum variable*. Since the set $\{\mathsf{c}_w \mid w \in \mathsf{W}_x\}$, for all memory locations x, is totally ordered with respect to $\prec$ in $\mathcal{T}_C$ by $\mathbf{WW}[x]$, the supremum variable sup_r evaluates to the maximum of all writes in W_x that happen-before r in R_x according to $\prec$. That is, $\mathsf{sup}_r = \mathsf{max}(\mathcal{H}_x(r))$ where $\mathcal{H}_x(r) \triangleq \{\mathsf{c}_w \colon \mathsf{c}_w \preceq \mathsf{c}_r$ and w is a write on $x\}$. By Theorem 4.2.14, for a given shared memory program structure P the formulas $\mathcal{E}^3$ and $\mathcal{E}^2$ are equisatisfiable.

Algorithm 6 Bitwise supremum encoding

Require: Let $x \in \langle ADDRESS \rangle$ be a memory location.
  for all read $r \in \mathsf{R}_x$ **do**
    $sup \leftarrow 0$ ▷ Zero fixed-size bit-vector
    for all write $w \in \mathsf{W}_x$ **do**
      $sup \leftarrow max(\mathsf{c}_w \;\&\; (0 - (\mathsf{c}_w \leq \mathsf{c}_r)) \;\&\; (0 - guard(w)), sup))$
  PRINT(sup)

4.4 Implementation and experiments

In this section, we describe various considerations when implementing the quadratic-size partial-order encoding shown in Figure 4.9 when $\mathcal{T}_C$, $\mathcal{T}_V$ and $\mathcal{T}_S$ can be assumed to be fixed-size bit-vector theories. We run our implementation on several benchmarks, which includes C code from the Apache HTTP server [AKT13], code from the SV-COMP `pthread` and `pthread-atomic` category [Bey13] and our OpenCores Ethernet benchmark from Chapter 2. We performed all experiments on a 64-bit Linux machine with eight Intel Xeon 3.07 GHz cores and 48 GB of main memory. For the SAT solver, we mainly used MiniSAT 2.2, except for the OpenCores Ethernet MAC benchmark for which we ran both MiniSAT 2.2 and Glucose 4.0 because we wanted to draw parallels to the earlier experiments in Chapter 2.

Since CBMC is originally designed as a SAT-based bounded model checker, it is reasonable to consider a bitwise encoding of $max(u, v) \triangleq (u + v + abs(u - v)) \gg 1$ where u and v are fixed-size bit vector variables of the same length with the usual two's complement semantics for fixed-size bit-vector addition ($+$), subtraction ($-$) and logical right-shifting without sign extension ($\gg$). The minimum number of required bits is $M = 1 + \lceil log_2(N) \rceil$ where N is the maximum number of writes. As pre-condition, we require that $u < 2^M$ as well as $v < 2^M$. Note that the extra bit is needed for sign extension because $u - v$ may result in a negative value. The *abs* function could be encoded using only bitwise operators.[3] Since this would require two 'XOR' operations, however, all current versions of CBMC encode the *abs* function using N if-then-else comparisons instead (see `bv_utilst::absolute_value` in `src/solvers/flattening/bv_utils.cpp`). Yet, another possible encoding of *max* could be obtained from fixed-size bit-vector equations such as $u - ((u - v) * (u \leq v))$ or $u + ((v - u) * (u \leq v))$ with the usual bitwise semantics for unsigned integer multiplication ($*$) and unsigned integer comparison ($\leq$). Given an encoding for *abs* or *max*, the $\mathbf{SUP}[x]$ constraint could be then encoded by Algorithm 6. For this to work, we interpret $\mathsf{c}_w \leq \mathsf{c}_r$

[3] http://graphics.stanfoyd.edu/~seander/bithacks.html#IntegerAbs

Thread $\mathtt{T}_1$	Thread $\mathtt{T}_2$	Thread $\mathtt{T}_3$		Thread $\mathtt{T}_N$
$[x_1] := 1$ $[y_1] := [x_2]$	$[x_2] := 1$ $[y_2] := [x_3]$	$[x_3] := 1$ $[y_3] := [x_4]$	$\cdots$	$[x_N] := 1$ $[y_N] := [x_1]$

Figure 4.10: A parametric version of the concurrent system in Figure A.1.

and $guard(w)$ as two separate bits which are zero-extended to the necessary width. Therefore, $0 - (\mathsf{c}_w \leq \mathsf{c}_r)$ is the fixed-size bit-vector whose bits are all '1' whenever $\mathsf{c}_w \leq \mathsf{c}_r$ is true, and '0' otherwise. Similarly for $0 - guard(w)$. Therefore, the bitwise logical 'AND' operator (&) forces the clock variable c_w to be zero if either $\mathsf{c}_w \leq \mathsf{c}_r$ or $guard(w)$ is false. One major drawback with such an implementation is that it is limited to SAT-based encodings of partial-order encodings, and therefore cannot benefit from the SMT-based features in CBMC.

To avoid this limitation, we encode the supremum in the standard mathematical way (Appendix B). The immediate benefit of such an encoding is that it makes our implementation less tightly coupled to SAT solvers because it can maintain a higher level of abstraction. This, in turn, makes other theory combinations conceivable, which is relevant for verification tasks where memory accesses could be abstracted through, say, an array theory. The required changes to CBMC 5.2 for this higher-level implementation are shown in Appendix E; it can be implemented in less than 100 lines of code after having settled on the internal representation of clock variables.

Setting aside for a moment the benefit of the higher-level implementation, we compared both approaches using a parameterized version of concurrent system in Figure A.1 where we changed register writes to shared memory writes. If we assume that all memory accesses are synchronized, Figure 4.10 sketches the resulting concurrent system where $1 < N$ is the parameter for the number of threads. We selected this benchmark for our comparison because the ratio between reads and writes on the same memory location remains constant as N varies, thereby fixing one important variable in this first experiment. For all $1 \leq i \leq N$, we assume that $[x_i]$ and $[y_i]$ are initialized to zero using a synchronized write. The safety property we want to check is the following: for all possible runs, after all N threads have terminated, there exists $1 \leq j \leq N$ such that $[y_j]$ is nonzero. As explained in Appendix A, the existence of such a j is guaranteed with interleaving semantics. With weak memory semantics such as TSO and PSO, however, it is possible that there exists a run such that $[y_j] = 0$ for all $1 \leq j \leq N$. Here, we use the PSO semantics as implemented by CBMC 4.5. Note that PSO in this case equates to SC-relaxed consistency where write/read pairs

Benchmark	$\mathcal{E}^3$				$\mathcal{E}^2$			
	Time	# SSA steps	# variables	# clauses	Time	# SSA steps	# variables	# clauses
fdqueue-ia32-safe	25 s	68,445	**830,913**	**4,250,727**	25 s	**8,584**	888,560	4,513,209
fdqueue-ppc-safe	24 s	59,955	**775,425**	**3,967,372**	23 s	**7,905**	834,106	4,223,630
builtin-safe	26 s	93,058	**999,842**	**5,022,551**	25 s	**10,311**	1,052,216	5,361,579

Table 4.1: Experimental results for the Apache HTTP server benchmark.

per memory location in the same thread are not included in the PPO. In our experimental comparison, we found that the implementation with the bitwise supremum encoding from Algorithm 6 times out after 15 *min* for $14 < N$, whereas the mathematical encoding from Appendix E completes in less than 4 *min* for all $N < 19$. Given the simplicity of the benchmark, we therefore avoid the bitwise supremum encoding, and use the mathematical encoding for our remaining experiments instead.

4.4.1 Apache HTTP and SV-COMP benchmarks

With this choice of reference implementation, we start with the same Apache HTTP server, and SV-COMP `pthread` and `pthread-atomic` benchmarks that were used in [AKT13]. In this experimental setup, if the run-time assertions in the C code cannot be violated, the benchmark is declared 'safe'; otherwise, it is called 'unsafe'. There are a total of 37 benchmarks, 21 of which are 'safe' and 16 are 'unsafe'. To check the assertions in the code, loops are unrolled a bounded number of times. As a sanity check, we have confirmed that both the cubic-size and quadratic-size encoding produce the expected result. Table 4.1 summarizes the experimental results for the Apache HTTP server benchmark. The columns report the following four metrics: the total run-time of CBMC in seconds, the number of steps that CBMC takes to construct the *single static assignment form* (*SSA*), as well as the number of required propositional variables and clauses of the SAT encoding. On average, the quadratic-size encoding decreases the number of SSA steps by a factor of eight. We highlight this benchmark because it draws attention to the fact that the decrease in SSA steps does not imply that the run-time and size of the SAT formulas necessarily decreases. In fact, the propositional encoding of $\mathcal{E}^2$ may require more propositional variables and clauses than the propositional encoding of $\mathcal{E}^3$. This can be explained by the fact that $\mathcal{E}^2$ introduces a new supremum variable per shared memory read.[4]

The run-time results for the SV-COMP `pthread` and `pthread-atomic` benchmarks are given in Figure 4.11. The two partial-order encodings are grouped per benchmark,

[4]As we shall see at the end of Chapter 5, introducing new variables can be in fact a good thing.

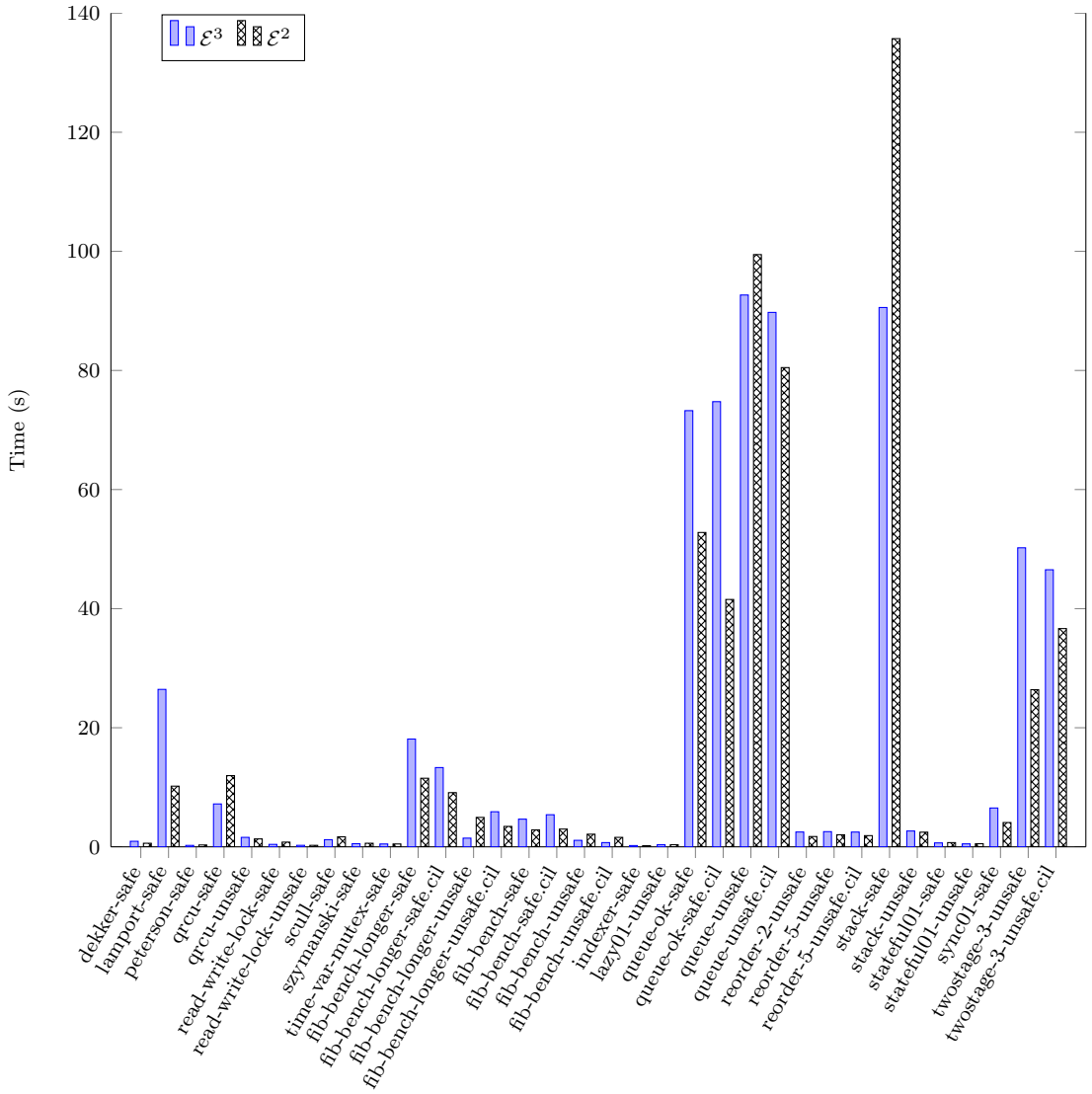


Figure 4.11: Experimental results for the SV-COMP `pthread` and `pthread-atomic` concurrency benchmarks [Bey13, AKT13].

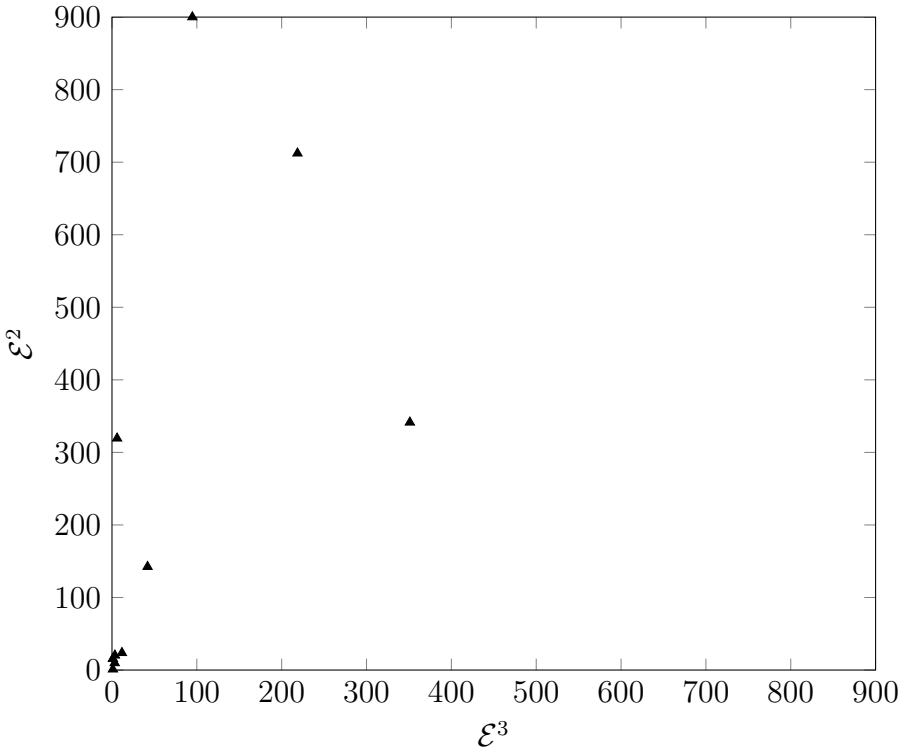


Figure 4.12: Scatter plot of analysis run-time in seconds when analyzing the concurrent system in Figure 4.10 using PSO semantics.

where the y-axis shows the total run-time of CBMC in seconds with a timeout of $900\,s = 15\,min$. For the `lamport-safe` and `queue-ok-safe` benchmarks, the quadratic-size speeds up CBMC by a factor of at least 1.8. On average, however, CBMC's run-time with the cubic-size encoding is 1.2 times faster compared to the quadratic-size encoding. This better average-case for the cubic-size encoding is largely due to the `stack-safe` benchmark in which CBMC is 1.5 times faster with the cubic-size encoding. We confirmed that the slowdown is due to the run-time of the decision procedure. Here, the cubic-size encoding yields 156,668 variables and 715,684 clauses, whereas the quadratic-size encoding requires slightly more, 158,892 variables and 719,223 clauses.

4.4.2 Parametric benchmark with fixed read/write ratio

To investigate this issue closer, we evaluated the cubic-size and quadratic-encoding with the previously discussed parametrized concurrent system in Figure 4.10 where the ratio of reads and writes is fixed. As explained, we rely on CBMC 4.5's implementation of the PSO semantics. The scatter plot in Figure 4.12 gives the run-time of the

	Cubic-size encoding				Quadratic-size encoding			
	Time (*min*)		SAT %	# SSA steps	Time (*min*)		SAT %	# SSA steps
	Total	SAT			Total	SAT		
	Glucose 4.0							
(HW.1)	**0.8**	0.6	70%	212,182	1.4	1.4	**99%**	**6,605**
(HW.2)	**26.4**	4.9	19%	2,273,594	125.2	125.0	**99%**	**60,177**
(HW.3)	**23.4**	2.2	9%	2,162,355	16.4	16.3	**99%**	**59,305**
(HW.4)	**28.3**	6.1	22%	2,273,905	62.5	62.3	**99%**	**61,353**
(HW.5)	**54.1**	10.8	20%	3,268,334	101.4	101.1	**99%**	**87,261**
(HW.1–5)	661.5	N/A	N/A	15,179,328	**14.5**	N/A	N/A	**459,673**
	MiniSAT 2.2							
(HW.1)	**0.7**	0.5	71%	212,182	1.5	1.5	**99%**	**6,605**
(HW.2)	32.1	10.9	34%	2,273,594	**22.3**	22.1	**99%**	**60,177**
(HW.3)	33.2	12.3	37%	2,162,355	**11.9**	11.7	**99%**	**59,305**
(HW.4)	36.7	14.8	40%	2,273,905	**22.4**	22.2	**99%**	**61,353**
(HW.5)	74.7	32.0	43%	3,268,334	**35.7**	35.4	**99%**	**87,261**
(HW.1–5)	651.7	N/A	N/A	15,179,328	**19.5**	N/A	N/A	**459,673**

Table 4.2: Comparison of the OpenCores Ethernet MAC benchmark results.

benchmarks. Here, the x-axis and y-axis plot the run-time with the cubic-size and quadratic-size encoding, respectively. On average, the run-time of CBMC with the quadratic-size encoding is at least one magnitude slower compared to the cubic-size encoding, despite the fact the number of SSA steps for the former is approximately 3 times less than that of the latter. In fact, in the case of $N = 17$, the quadratic-size encoding causes CBMC to timeout, whereas CBMC with the cubic-size encoding successfully runs to completion in less than 2 *min*, confirming that the decision procedure dominates the total run-time in this parametric benchmark.

Our experiments with the parametric benchmark poses the question when the trade-off in the number of SSA steps and solver efficiency is worth it. In the next subsection, we answer this question using a more sophisticated benchmark.

4.4.3 Revisited: OpenCores Ethernet MAC benchmark

The previous experiments give a good idea when the quadratic-size partial-order encoding provides no real advantages. This includes relatively small benchmarks that emit only a small number of SSA steps. In this subsection, we revisit the OpenCores Ethernet MAC benchmark from Chapter 2 that initially prompted the quest for a smaller partial-order encoding. The experimental setup is the same as in Section 2 except that we modify CBMC 5.2 to use the quadratic-size partial-order encoding.

Recall that the required changes to CBMC 5.2 are described in Appendix E.

Table 4.2 compares our new experimental results for the OpenCores Ethernet MAC benchmark to the previous ones from Section 2.4.[5] As already observed in the experiments from the previous subsections, the CNF formula of the quadratic-size partial-order encoding has more variables and clauses compared to the cubic-size encoding. However, for all benchmarks except (HW.1), the memory consumption of the quadratic-size partial-order encoding is around 500 MB lower (using either Glucose and MiniSAT) compared to the cubic-size partial-order encoding. More importantly, irrespective of the choice of SAT solver, the new encoding reduces the number of SSA steps by at least one order of magnitude. This, in turn, means that nearly 99% of the total run-time is now spent in the SAT solver—a huge improvement over the cubic-size partial-order encoding where only at most 30–40% of the total run-time was used for SAT solving. In the case of MiniSAT (provided no MEMOUT error occurs within the SAT solver), this measurably speeds up the total run-time despite the fact that the quadratic-size partial-order encoding noticeably slows down the SAT solver. Surprisingly, in the case of Glucose, this slow down is greater than the time that was saved by reducing the steps in CBMC for constructing the partial-order encoding. This is surprising because Glucose outperformed MiniSAT on the cubic-size partial-order encoding (cf. Table 2.1), thereby reversing the trends for the total run-time that we observed earlier in Section 2.4. Table 4.3 links this to more detailed SAT solving statistics, showing that in the majority of metrics Glucose needs to make significantly more decisions and encounters considerably more conflicts than MiniSAT.[6] The comparison of the MiniSAT statistics for both encodings shows that the SAT solving times are stable but the quadratic-size encoding leads to a measurable increase in the number of restarts, decisions, conflicts and learnt literals. This may explain why 'HW.1–5' can also not be solved with the quadratic-size partial-order encoding. More importantly, however, the new 'HW.1–5' results show a decrease in run-time from around 10 hours to less than 20 minutes for both MiniSAT and Glucose—a one order of magnitude speedup. This reinforces the conclusion of our in-depth analysis of the cubic-size encoding for this benchmark (cf. Section 2.4) where we showed that most of the 10 hours is spent in CBMC rather than the SAT solver.

We have also repeated the experiment by using Lingeling by Armin Biere because it is one of the SAT solvers that is not based on MiniSAT. For this, we exported the benchmarks into the DIMACS format. We found that Lingeling runs on average

[5] The details can be found in Table F.1.

[6] The details of this SAT solver comparison can be found in Table F.2.

	Ratio of SAT statistics: quadratic/cubic					
	Glucose 4.0					
Property	SAT Time	Starts	Decisions	Propagations	Conflicts	Learnt Literals
(HW.1)	2.6 X	152.5 X	157.9 X	341.0 X	86.9 X	53.8 X
(HW.2)	30.2 X	17.1 X	19.1 X	40.7 X	16.9 X	5.9 X
(HW.3)	9.5 X	3.5 X	21.5 X	20.2 X	9.1 X	5.7 X
(HW.4)	21.1 X	7.3 X	7.2 X	26.6 X	18.8 X	1.6 X
(HW.5)	10.5 X	10.6 X	36.3 X	12.4 X	7.4 X	2.4 X
	MiniSAT 2.2					
(HW.1)	2.6 X	35.1 X	269.6 X	226.4 X	84.0 X	434.1 X
(HW.2)	2.0 X	4.0 X	6.8 X	10.6 X	5.1 X	1.4 X
(HW.3)	1.0 X	3.0 X	7.4 X	7.8 X	3.9 X	5.3 X
(HW.4)	1.6 X	3.4 X	9.8 X	8.2 X	4.1 X	2.6 X
(HW.5)	1.0 X	3.0 X	10.9 X	5.7 X	3.3 X	1.0 X

Table 4.3: Ratio of SAT solver statistics for OpenCores Ethernet MAC benchmark.

approximately 7 X times slower on the quadratic-size partial-order encoding compared to the cubic-size partial-order encoding.[7]

Since the only constraints that were added to the quadratic-size partial-order encoding are the supremum variables, we set up a last experiment in which we change the encoding of the supremum variables (see Section 4.3) such that they are translated to non-decision variables in the propositional formula.[8] This means that the literals that encode the supremum variables are ignored by the branching heuristics of the SAT solver. For both MiniSAT and Glucose, this speeds up the SAT solving time by a small constant factor less than 2.[9] In fact, in the case of (HW.5), the number of decisions drops by one order of magnitude but the number of conflicts stays almost the same. This suggests that the slowdown in Glucose cannot be attributed to superfluous decisions on the bit-blasted supremum variables.

In summary, our experimental results provide strong evidence that the quadratic-size partial-order encoding slows down SAT solvers. We showed that this slowdown cannot be attributed to superfluous decisions on the bit-blasted supremum variables. What is remarkable is that the quadratic-size partial-order encoding can still significantly speedup the total run-time of the analysis when the program under scrutiny is sufficiently large. This is due to the fact that the construction of the cubic-size partial-order encoding can take longer than actually solving it.

[7] https://github.com/ahorn/benchmarks/commit/5a7034a8d95397a906f74eb8301147e8ec
[8] https://github.com/ahorn/benchmarks/commit/c580d71e07b6c105e2dc242bc8fc4ddaaf
[9] The full experimental results are given in Table F.3.

4.5 Concluding remarks

We defined SC-relaxed consistency by further restricting the downward-closure of programs from Chapter 3. Using SC-relaxed consistency, we showed how data races can give rise to N-shaped partial strings, whose importance has been recognized in the theoretical computer science literature for a different reason [Pra86]. More generally, SC-relaxed consistency can model weak memory semantics such as PSO and TSO by relaxing the PPO. Since these relaxations are highly non-trivial in the presence of pointer dereferences, we carefully designed our implementation such that it can leverage the PPO computed by CBMC.

Our experimental comparison with the same reference implementation and a range of benchmarks show that the performance of a partial-order encoding cannot be predicted by the number of partial-order constraints alone, even though it is widely used as a main metric for comparison [AKT13]. This is witnessed by two benchmarks in particular. The first of these we purposefully devised such that the ratio of reads and writes on the same memory location remained constant. This made it possible to eliminate further variables from the experiment while also amplifying the performance differences between both partial-order encodings. The second benchmark with the OpenCores Ethernet MAC hardware model is more realistic and shows that the quadratic-size partial-order encoding can nevertheless significantly speedup the total run-time of the analysis when the program under scrutiny is sufficiently large. We have shown, however, that this speedup strongly depends on the SAT solver.

Would a first-order logic encoding achieve better results? This could be answered with a better understanding on how partial-order encodings interact with the underlying decision procedure. For this purpose, the next chapter gives experiments with a parametric concurrency problem for which the number of SAT conflicts is shown to increase exponentially in the number of threads, irrespective of the choice of propositional and first-order logic (including the theory combination $\mathcal{T}_C + \mathcal{T}_V + \mathcal{T}_S$), or the quadratic and cubic partial-order encoding.

4.6 Bibliographic notes

Recently much research has gone into formalizing language-specific (e.g. [BOS+11]) or architecture-specific (e.g. [SSO+10, vVZN+11, SSA+11, AMSS12, MHMS+12]) weak memory concurrency. The partial-order encodings in this chapter are neither language-specific nor architecture-specific but rather focus on how to leverage off-the-

shelf, highly optimized decision procedures for solving NP-hard problems.

Early on, bounded model checking techniques emerged that symbolically encode all interleavings (or a subset thereof) in the form of symbolic context switches, e.g. [RG05, GG08, LQR09]. This includes more recent symbolic sequentialization techniques, e.g. [ITF+14]. Our work is part of this line of research except that we are investigating a symbolic technique that can be naturally extended to non-interleaving semantics. The state-of-the-art symbolic technique for achieving this is the partial-order encoding in CBMC [AKT13], which is the most closely related work to ours by Theorem 4.2.14.

Related work that predates CBMC's partial-order encoding includes [YGLS03, YGLS04, GYS04, BAM07, WKGG09, TVD10, SW10, BWB+11, MHMS+12]. Among these, Nemos [YGLS03, YGLS04] is one of the earliest automated software verification tools that axiomatically specifies weaker forms of consistency using decidable fragments of higher-order logic order constraints. To solve those constraint automatically, Nemos is implemented as a Prolog program that is defined recursively over finite data types, including a two-dimensional matrix data structure that represents a binary relation. Since this implementation does not support branch or comparison instructions, it would not be able to directly check simple concurrent system such as shown in Figure 4.8. Follow-up work [GYS04] on the Intel Itanium memory model improves on the previous SAT-based approach [YGLS03] by carefully tuning the Boolean encoding. While the authors found that a matrix encoding using a quadratic number of Boolean literals could be solved more efficiently by the decision procedure compared to the more compact logarithmic encoding, they note that a litmus test of 130 assembly instructions could not be handled by the constraint generator due to the large size of the matrix encoding.[10] A similar matrix encoding is used by CheckFence [BAM07] with one main novelty: CheckFence encodes branch statements. The importance of encoding branch statements for the purpose of finding concurrency-related bugs is effectively demonstrated by later work on the Fusion tool [WKGG09]. Unlike CheckFence, Fusion encodes the partial-order constraints into difference logic using an SMT solver. This proved successful on sequentially consistent programs for which dynamic partial-order reduction techniques would not scale well [WKGG09]. Independently, both [TVD10, SW10] further improve on this idea by handling acyclic control-flow graphs that are translated to a variant of SSA in which every definition of a variable is given a new name. MemSAT [TVD10] and CppMem [BWB+11] are similar in that both can check litmus tests by automatically encoding axiomatic

[10] A similar impediment was motivation for devising our quadratic-encoding for CBMC.

specifications of memory models using relations.

In Example 4.2.2 and 4.3.4, we explained the notion of data races in the context of partial strings. We could automatically detect such data races by adding to our partial-order encoding clock constraints of the form $\mathsf{c}_w = \mathsf{c}_{w'}$ or $\mathsf{c}_w = \mathsf{c}_r$ in $\mathcal{T}_C$ where $w \neq w'$ and r are write and read events on the same memory location, respectively. But the literature offers many other (possibly more efficient) techniques for automatically detecting data race. These can be divided dynamic and static techniques. We primarily focus our discussion on statically detecting data races in weaker forms of memory rather than sequentially consistent systems, e.g. [GG08]. Dynamic data race detection techniques for weak memory architectures are polynomial time efficient but necessarily imprecise, e.g. [AHMN91, GG91, SI09, PPH+11, DWS+12, XXZ13]. Recent tools can statically detect data races for weak memory programs using explicit model checking [KYKS12] or by explicitly enumerating happens-before relations [BOS+11, ND13]. This includes most recently developed tools in the herd family [AMT14].[11] One of the earliest symbolic data race detection tools [YGL04] is based on Nemos [YGLS04] (see earlier discussion).

Elementary programs (Definition 3.3.9) are a generalization of the 'neighbourhood' universe of discourse in [GHR+15] which is limited to the downward-closure with respect to per-thread event orderings. An HB-formula [GHR+15] can represent arbitrary subsets of neighborhoods and may not therefore be an elementary program.

[11] https://github.com/herd/herdtools

Chapter 5

Limitations of DPLL($\mathcal{T}$) for SMT-based partial-order encodings

Collaborators *The bulk of the material in this chapter comes from [HHK15], authored in collaboration with Liana Hadarean and Tim King. My contributions to this paper are listed in Section 1.1.*

5.1 Introduction

As we have seen in Section 4.3, partial-order encodings are inherently structured around three separate theories: $\mathcal{T}_C$, $\mathcal{T}_V$ and $\mathcal{T}_S$. It is therefore natural to ask how this structure may be exploited for more efficient automated verification of shared memory programs. So far, the relevance of this question may not have been so clear because we only considered SAT-based partial-order encodings of shared memory concurrency, where most of the structure appears to get lost due to the limited expressiveness of propositional logic.

To retain more of the structure in partial-order encodings, we turn to decidable fragments of first-order logic instead. A large class of decidable first-order logic fragments is formalized by the SMT-LIB standard [BFT15]. This standard allows us to express many practical concurrency problems with mathematical precision yet also in a machine-readable format that we can feed to SMT solvers. Our goal is to investigate how the structure of partial-order encodings affects the decision procedures underpinning SMT solvers.

At their core, most state-of-the-art SMT solvers implement a variant of the so-called DPLL($\mathcal{T}$) framework, a form of modular conflict-driven backtracking search that supports multiple theory-specific decision procedures. To achieve this modular-

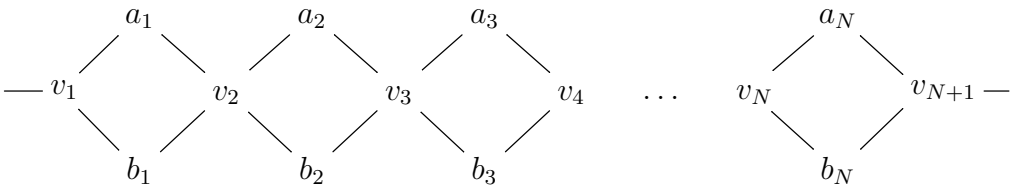
ity, the DPLL($\mathcal{T}$) framework consists of two parts: (i) a SAT solver, and (ii) one or more theory-specific solvers. The former (i) typically implements a form of *Conflict Driven Clause Learning* (*CDCL*), which combines aspects of the classical DPLL algorithm [DP60, DLL62] with a form of learning [MSS96, BS97, MSS99]. The latter (ii) is designed to check whether a conjunction of theory literals is satisfiable.

For simplicity, we fix $\mathcal{T}$ to be a single theory. To decide the satisfiability of a first-order logic formula in the theory $\mathcal{T}$, both solvers need to interact with each other. Informally, this interaction works as follows. The SAT solver searches for a satisfying assignment to the propositional abstraction of the input formula. When such an assignment is found, the $\mathcal{T}$-solver checks whether this propositional assignment is $\mathcal{T}$*-consistent*. If it is, the input formula is satisfiable in the theory $\mathcal{T}$, and the SMT solver terminates. Otherwise, we get a $\mathcal{T}$*-conflict* clause that can be seen as a logical summary of the theory inconsistency. The negation of a $\mathcal{T}$-conflict is a tautology, and therefore called a $\mathcal{T}$*-lemma*. Such a theory lemma, for each $\mathcal{T}$-conflict that is encountered during the search, is added to the input formula in order to prevent the SAT solver from exploring this part of the search space again. This process is a form of 'learning' that continues until either a theory-consistent satisfying assignment is found, or a contradiction can be derived by the SAT solver.

While usually efficient in practice, there are well-known problems, such as the "diamonds problem" [SSB02], for which the DPLL($\mathcal{T}$) framework cannot derive a contradiction by learning a polynomial number of theory lemmas. The *diamonds problem* is the following formula in the theory of uninterpreted functions and equality for some fixed constant N:

$$\phi_\diamond \triangleq \left(\bigwedge_{i=1}^{N} (v_i = a_i \wedge a_i = v_{i+1}) \vee (v_i = b_i \wedge b_i = v_{i+1}) \right) \wedge v_1 \neq v_{N+1}.$$

Note that $\phi_\diamond$ is unsatisfiable because $v_i = v_{i+1}$ for all $1 \leq i \leq N$. Diagrammatically, the formula can be visualized by the following graph in the shape of N diamonds:



Each edge in the above graph represents an equality and the two paths from v_i to v_{i+1} correspond to two disjuncts. DPLL($\mathcal{T}$) enumerates 2^N theory conflicts before it

can derive a proof that $\phi_\diamond$ is unsatisfiable. Note that each of these theory conflicts corresponds to one path in the above graph. It turns out that this exponential explosion in theory conflicts stems from the fixed alphabet of theory literals. Despite work on addressing this inherent inefficiency, finding an efficient general purpose method is still an open problem [BDdM08, TR12].

We show that a variation of the "diamonds problem" resurfaces in partial-order encodings of SC-relaxed consistency. To show this, we study a simple concurrency problem that is parametric in the number of threads, each of which accesses a single shared memory location. Our concurrency problem is therefore minimalistic; it only requires the 'SC-per-memory-location' law [AMT14]. Based on this concurrency problem, we give new theoretical and experimental DPLL($\mathcal{T}$) proof complexity results. Our experiments feature the cubic-size and quadratic-size partial-order encoding from Section 4.3. Moreover, for experimental robustness, we run all experiments with four SMT solvers (Boolector, CVC4, Yices, Z3) and four SMT-LIB theory combinations. Our experiments take the form of standalone SMT-LIB benchmarks. We propose a total of 56 such benchmarks, which we refer to collectively as the *concurrency challenge problem*. Our experimental results using the concurrency challenge problem show that the number of generated theory conflicts on these benchmarks grows at a factorial rate in the number of writes on the shared memory location, irrespective of the choice of SMT solver and combination of SMT-LIB theories. Our experimental setup is the first of its kind to isolate this issue and draw attention to the problem in the context of partial-order encodings. More importantly, our experiments pinpoint one possible root cause of the performance bottleneck. To address this issue, we show how the majority of our benchmarks can be efficiently solved by most SMT solvers by adding certain interval theory lemmas to the input formula.

Contributions The main contributions of this chapter are as follows:

1. We give a new result for establishing lower bounds on the size of DPLL($\mathcal{T}$) proofs of unsatisfiability. This result is significant because it provides a diagnostic tool for SMT-LIB encodings.

2. Using our theory, we establish a factorial lower bound on the size of DPLL($\mathcal{T}$) proofs of unsatisfiability of our concurrency challenge problem, whose solution is directly relevant to finding concurrency-related bugs. This provides strong evidence for the obstacles that the current DPLL($\mathcal{T}$) framework poses for efficiently exploiting the structure in partial-order encodings.

Thread $\mathtt{T}_0$		Thread $\mathtt{T}_1$			Thread $\mathtt{T}_N$
local $v_0 := [x]$ **assert**$(v_0 \leq N)$	$\Vert$	**local** $v_1 := [x]$ $[x] := v_1 + 1$	$\Vert$	$\cdots$	$\Vert$ **local** $v_N := [x]$ $[x] := v_N + 1$

Figure 5.1: A concurrent system $\mathtt{T}_0 \parallel \mathtt{T}_1 \parallel \ldots \parallel \mathtt{T}_N$ where $0 < N$ is a fixed integer. We want to check the assertion in thread $\mathtt{T}_0$, assuming $[x]$ is initially 0.

3. We experimentally confirm the hardness of our proposed concurrency challenge problem. Moreover, our experiments reveal that the number of SAT conflicts grows exponentially faster when bit-blasting. This rules out current SAT solver technology as an easy fix to our problem.

4. We show how the problem encoded into integer arithmetic could be efficiently solved by adding theory lemmas in the form of unbounded intervals, providing an approach for improving the scalability of SMT-based partial-order encodings.

Organization The rest of this chapter is organized as follows. We first describe the concurrency challenge problem (Section 5.2). We then formalize what we mean by Fixed-Alphabet DPLL($\mathcal{T}$) proofs (Section 5.3). This formalism is the basis for our lower bounds results (Section 5.4 and 5.5). We experimentally confirm the factorial lower bound for the concurrency challenge problem (Section 5.6). Finally, we show how a simplified version of the problem could be solved efficiently (Section 5.7).

5.2 Concurrency challenge problem

In this section, we present a challenge problem based on the `fpk2013` SV-COMP concurrency benchmark [FKP]. This benchmark, in turn, is inspired by a problem that was first introduced in 1976 to illustrate the need for auxiliary variables in compositional proof rules for concurrent programs [OG76]. Recently, this problem has reappeared in the literature on automated verification tools [FKP13].

Consider the simple concurrent system in Figure 5.1 which features $N+1$ threads that access a single shared memory location x. While we could assume that all memory accesses to x are synchronized (see Chapter 4), for simplicity we require sequential consistency per memory location instead. As remarked in Section 5.1, this is still not an overly restrictive requirement. Recall that the memory at location x is denoted by $[x]$. We assume that $[x]$ is initially zero. Each thread $\mathtt{T}_i$ reads the value at memory location x into a CPU-local register v_i. For $i \geq 1$, thread $\mathtt{T}_i$ overwrites

the memory at location x with the new value $v_i + 1$. For the rest of the chapter, we denote the read of memory location x in $\mathtt{T}_0$ by r_{assert}. The read and write on memory location x in thread $\mathtt{T}_i$ for $i \geq 1$ are denoted by r_i and w_i, respectively. This gives us the following abstract rendition of Figure 5.1:

Thread $\mathtt{T}_0$	Thread $\mathtt{T}_1$		Thread $\mathtt{T}_N$
r_{assert}	r_1	$\ldots$	r_N
	w_1		w_N

By assumption that each memory access on x is sequentially consistent, we could enumerate all interleavings of these read and write events. For example, if we just consider the concurrent system $\mathtt{T}_1 \parallel \mathtt{T}_2$, we get the following six interleavings of shared memory accesses:

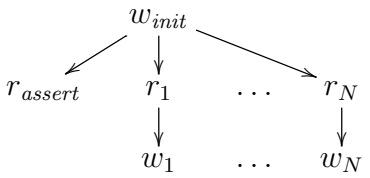
$$(1)\ r_1; w_1; r_2; w_2 \quad (2)\ r_1; r_2; w_1; w_2 \quad (3)\ r_1; r_2; w_2; w_1$$
$$(4)\ r_2; r_1; w_1; w_2 \quad (5)\ r_2; r_1; w_2; w_1 \quad (6)\ r_2; w_2; r_1; w_1$$

The different interleavings can result in different final values of $[x]$. For example, the first interleaving, $r_1; w_1; r_2; w_2$, results in the final value 2 at memory location x, whereas the second interleaving, $r_1; r_2; w_1; w_2$, results in the final value $[x] = 1$.

We want to check that the assertion $v_0 \leq N$ in thread $\mathtt{T}_0$ cannot be violated in the concurrent system $\mathtt{T}_1 \parallel \ldots \parallel \mathtt{T}_N$. Intuitively, this assertion holds because each of the other N threads increments $[x]$ at most once. For a fixed N, we want to prove this automatically using bounded model checking. While it is easy to automatically prove this property on each separate interleaving, the number of interleavings grows exponentially ($(2N + 1)! \div 2^N$).

Next, we explain how to generate partial-order encodings that formalize all interleavings as a single quantifier-free SMT query. For this, we fix a shared memory program structure (recall Definition 4.3.1). For $N = 2$, this structure looks as follows:

- The set of events $E = \{w_{init}, r_1, w_1, r_2, w_2, r_{assert}\}$ is partitioned into read events, $\mathsf{R}_x = \{r_1, r_2, r_{assert}\}$, and write events, $\mathsf{W}_x = \{w_{init}, w_1, w_2\}$, where x denotes the concrete memory location accessed by threads $\mathtt{T}_0$, $\mathtt{T}_1$ and $\mathtt{T}_2$.
- By sequential consistency, the PPO is as follows:



  Therefore, for $N = 2$, $w_{init} \ll r_{assert}$, $w_{init} \ll r_1 \ll w_1$ and $w_{init} \ll r_2 \ll w_2$.

- The $val : E \rightarrow \mathcal{T}_V$-terms function is defined as $val(w_{init}) \triangleq 0$, $val(w_1) \triangleq \mathsf{rv}_{r_1} + 1$ and $val(w_2) \triangleq \mathsf{rv}_{r_2} + 1$ where rv_{r_1} and rv_{r_2} are the read variables (recall Definition 4.3.3) for the read events r_1 and r_2, respectively.

- Finally, since the concurrent system has no if-then-else statements, $guard(e) =$ '**true**' for all events e in E.

It is easy to generalize this shared memory program structure to $N + 1$ threads. We begin by constructing a formula from the above shared memory program structure that encodes the challenge problem using the $\mathcal{E}^3$ encoding. Recall that the intuition behind $\mathcal{E}^3$ is as follows. Inter-thread accesses to shared state are first decoupled by being given distinct symbolic names. Furthermore, a clock symbol is introduced for each access to shared state, and a partial-order is given among the clocks according to some memory semantics (here, it is sequential consistency). Finally, order-dependent equalities are established among the decoupled symbolic names, making connections between the state values 'seen' by the threads.

Since it will be later convenient for $\mathcal{E}^3$ to be in CNF, we perform the following simplifications: (i) all of the guards, for all events in E, are ignored because they always evaluate to **true**, and (ii) implications are distributed across conjunctions in the $\mathbf{RF}^3[x]$ constraints, i.e. $[A \Rightarrow (B \wedge C)]$ is equivalent to $(A \Rightarrow B) \wedge (A \Rightarrow C)]$. We assume that the theory of value constraints $\mathcal{T}_V$ (see Section 4.3) is either bit-vector, Presburger, or real arithmetic. Recall that $\mathcal{T}_C$ and $\mathcal{T}_S$ denote clock and selection constraints, respectively, and $\mathcal{T}$ denotes the combined theory $\mathcal{T}_C + \mathcal{T}_V + \mathcal{T}_S$. Figure 5.2 shows the quantifier-free $\mathcal{T}$-formula, denoted by ϕ^3, that encodes the concurrency challenge problem. Note that ϕ^3 is in CNF if we interpret implications in the obvious way. Furthermore, note that in the $\mathbf{RF}^3[x]$ constraints, each $val(w)$ term has been replaced by either 0 or $\mathsf{rv}_{r_i} + 1$. By assuming $\prec$ is total, we can further simplify the $\mathbf{WW}[x]$ and $\mathbf{RW}[x]$ constraints as follows:

$$\underbrace{\bigwedge_{w,w' \in \mathsf{W}, w \neq w'} \mathsf{c}_w \neq \mathsf{c}_{w'} \wedge \mathsf{s}_w \neq \mathsf{s}_{w'}}_{\mathbf{WW}[x]} \qquad \underbrace{\bigwedge_{w \in \mathsf{W}, r \in \mathsf{R}} \mathsf{c}_w \neq \mathsf{c}_r}_{\mathbf{RW}[x]}$$

Similarly, the corresponding formula using the quadratic partial-order encoding $\mathcal{E}^2$ is also straightforward to derive from Figure 4.9. Since we want to reason about both $\mathcal{E}^3$ and $\mathcal{E}^2$ in terms of the DPLL($\mathcal{T}$) framework, we first formalize DPLL($\mathcal{T}$) abstractly, hiding the implementation details of SMT solvers.

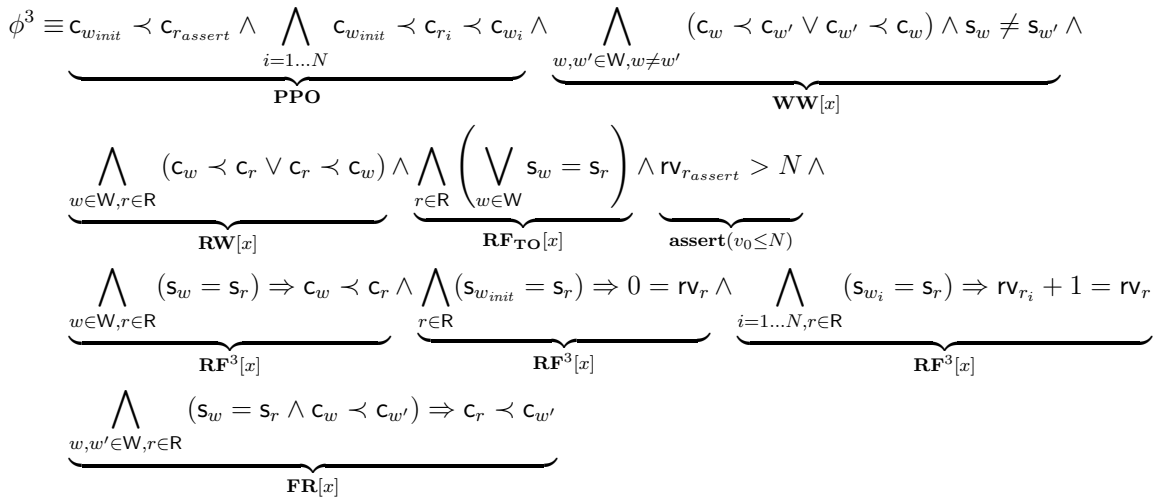


Figure 5.2: The $\mathcal{E}^3$ encoding for the challenge problem.

5.3 Fixed-alphabet DPLL($\mathcal{T}$)

In this section, we fix a simplified view of DPLL($\mathcal{T}$) that allows us to reason about the size of DPLL($\mathcal{T}$) proofs of unsatisfiability. Before doing so, we illustrate the concrete DPLL($\mathcal{T}$) algorithm (how it works operationally) through the following example.

Example 5.3.1. Consider the first-order logic formula $\phi \triangleq (x < y \vee x = y) \wedge y < x$ where x and y are integer variables with the usual meaning for strict inequality ($<$) and equality ($=$). Here is a potential sequence of DPLL($\mathcal{T}$) steps for proving that ϕ is unsatisfiable. Foremost, note that the formula consists of the following three $\mathcal{T}$-atoms: $x < y$, $x = y$ and $y < x$. Let $\{\ell_1, \ell_2, \ell_3\}$ be a set of propositional variables. We can then associate each of these $\mathcal{T}$-atoms with a unique propositional variable, say by the injective mapping $\{x < y \mapsto \ell_1, x = y \mapsto \ell_2, y < x \mapsto \ell_3\}$. In the first iteration, suppose the SAT solver finds the following satisfying assignment $M_1 = \{\ell_1, \neg\ell_2, \ell_3\}$. That is to say, the propositional variable for $x < y$ and $y < x$ is assigned 'true', whereas the propositional variable for $x = y$ is assigned 'false'. Given this satisfying assignment, the theory solver finds the $\mathcal{T}$-conflict $x < y \wedge y < x$, whose negation is the following $\mathcal{T}$-lemma: $\neg(x < y) \vee \neg(y < x)$. By using the above injective mapping, the propositional abstraction of this $\mathcal{T}$-lemma is communicated back to the SAT solver via $\neg\ell_1 \vee \neg\ell_3$. The SAT solver continues and finds another satisfying assignment, say $M_2 = \{\neg\ell_1, \ell_2, \ell_3\}$. This leads to a second theory conflict, namely $x = y \wedge y < x$. The corresponding propositional abstraction of the theory lemma, $\neg\ell_2 \vee \neg\ell_3$, is added to

the SAT solver. The previous two steps result in the following CNF formula:

$$(\ell_1 \vee \ell_2) \wedge \ell_3 \wedge (\neg\ell_1 \vee \neg\ell_3) \wedge (\neg\ell_2 \vee \neg\ell_3).$$

At this point, the SAT solver can conclude that it is unsatisfiable on the propositional level. Since only theory valid clauses were added during the search, the theory solver can therefore conclude that the input formula ϕ is unsatisfiable.

We want to describe abstractly how the propositional and theory-specific reasoning interact in proving the unsatisfiability of a formula. For this purpose, we consider a simplified view of the DPLL($\mathcal{T}$) calculus [NOT06] with only two rules:

- Propositional resolution (Res);
- Learning of the propositional abstraction of $\mathcal{T}$-lemmas over the literals of a fixed alphabet of $\mathcal{T}$-atoms ($\mathcal{T}$-learn).

The idea behind both rules is to allow DPLL($\mathcal{T}$) proofs of unsatisfiability that can consist of a combination of learning $\mathcal{T}$-lemmas and resolution steps on their propositional abstraction including the input formula, until the empty clause is derived.[1] As in [NOT06], we restrict those DPLL($\mathcal{T}$) proofs to be over the fixed alphabet $\mathcal{A}$ of $\mathcal{T}$-atoms in the input formula. Moreover, we assume that all $\mathcal{T}$-lemmas are clauses. We ignore $\mathcal{T}$-propagation and splitting-on-demand [BNOT06] because it can introduce new $\mathcal{T}$-atoms. For example, a splitting-on-demand $\mathcal{T}$-solver for a decidable fragment of set theory may reduce the disequality $S \neq S'$ of two sets S and S' as the following disjunction [BNOT06]: $(c \in S \wedge c \notin S') \vee (c \notin S \wedge c \in S')$ where c is a fresh Skolem constant. Our simplified form of DPLL($\mathcal{T}$) is formalized as follows.

We fix a set of propositional variables $\mathcal{X}$. Let $\mathcal{L}_\mathcal{X}$ be the set of literals over $\mathcal{X}$ where a *literal* is either a propositional variable in $\mathcal{X}$ or its negation. We denote a literal in $\mathcal{L}_\mathcal{X}$ by ℓ. A clause C is a set of literals interpreted as their disjunction. The empty clause $\square$ denotes false. The negation of a clause is a set of literals $\neg C = \{\neg\ell \mid \ell \in C\}$ and is interpreted as a conjunction.

The propositional abstraction function $\_^{\mathbb{B}}$ is an injective map from the fixed-alphabet $\mathcal{A}$ into the set of propositional variables $\mathcal{X}$. The $\mathcal{T}$-literals, written $\mathcal{L}_\mathcal{A}$, are the set of literals over $\mathcal{A}$. We lift $\_^{\mathbb{B}}$ to work over $\mathcal{T}$-literals and sets thereof. We denote by L a $\mathcal{T}$-valid clause over $\mathcal{L}_\mathcal{A}$, $\models_\mathcal{T} \bigvee_{t\in L} t$, and $\neg L$ will denote a $\mathcal{T}$-conflict. So a $\mathcal{T}$-conflict is a set of $\mathcal{T}$-literals whose conjunction is $\mathcal{T}$-unsatisfiable, $\neg L \models_\mathcal{T} \square$.

[1] This still allows for other reasoning steps such as theory propagation.

We assume the input $\mathcal{T}$-formula ϕ has already been converted to CNF and is represented as a finite set of clauses $C_1, \ldots, C_\alpha$ over the propositional variables in $\mathcal{X}$, the set of $\mathcal{T}$-atoms $\mathcal{A}$, together with the lifted boolean abstraction function $\_^{\mathbb{B}}$ that formalizes the one-to-one mapping between $\mathcal{T}$-literals and propositional literals. A Fixed-Alphabet DPLL($\mathcal{T}$) proof has the following form:

$$C_1, \ldots, C_\alpha, \ldots, C_k, \ldots, C_\beta = \square$$

where each C_k for $\alpha < k \leq \beta$ is a clause over $\mathcal{L}_\mathcal{X}$ that is derived from a previous clause using either the resolution rule (Res) or theory learning ($\mathcal{T}$-learn). We detail each rule in turn.

The rule $\mathcal{T}$-learn adds a new clause $L^{\mathbb{B}}$ that corresponds to the propositional abstraction of a $\mathcal{T}$-valid clause L over the $\mathcal{T}$-literals in $\mathcal{L}_\mathcal{A}$. Formally,

$$\frac{C_1, \ldots, C_k \quad L \subseteq \mathcal{L}_\mathcal{A} \quad \models_\mathcal{T} \bigvee_{t \in L} t}{C_1, \ldots, C_k, L^{\mathbb{B}}} \ \mathcal{T}\text{-learn}.$$

The clauses derived by $\mathcal{T}$-learn are called $\mathcal{T}$-*lemmas*. Note that $\mathcal{T}$-learn is more general than Lazy Theory Learning [NOT06], which requires the literals to be in the partial assignment (see also Section 5.4).

For the Res rule, let $C_i \otimes_\ell C_j$ denote propositional resolution on ℓ, and define Res as follows:

$$\frac{C_1, \ldots, C_k \quad 1 \leq i < j \leq k \quad \ell \in C_i \quad \neg\ell \in C_j}{C_1, \ldots, C_k, C_i \otimes_\ell C_j} \ \text{Res}.$$

The Res rule is refutation-complete which means SAT solver unsatisfiability proofs can be explained in terms of propositional resolution. To illustrate Fixed-Alphabet DPLL($\mathcal{T}$) proofs, we recast Example 5.3.1 into our formalism.

Example 5.3.2. Given the input formula ϕ in Example 5.3.1, define the set of propositional variables and $\mathcal{T}$-atoms to be $\mathcal{X} = \{\ell_1, \ell_2, \ell_3\}$ and $\mathcal{A} = \{x < y, x = y, y < x\}$, respectively. The propositional abstraction function is defined as follows:

$$\_^{\mathbb{B}} \triangleq \{x < y \mapsto \ell_1, x = y \mapsto \ell_2, y < x \mapsto \ell_3\}.$$

This injection $\_^{\mathbb{B}} \colon \mathcal{A} \to \mathcal{X}$ is lifted to $\_^{\mathbb{B}} \colon \mathcal{L}_\mathcal{A} \to \mathcal{L}_\mathcal{X}$ in the natural way.

The input formula ϕ is represented by the following two clauses $C_1 = \{\ell_1, \ell_2\}$ and $C_2 = \{\ell_3\}$. Let $L_3 = \{\neg(x < y), \neg(y < x)\}$ be a subset of $\mathcal{L}_\mathcal{A}$. In the theory of

linear integer arithmetic, $\models_{\mathcal{T}} \neg(x < y) \vee \neg(y < x)$ is $\mathcal{T}$-valid because the integers are totally ordered by $\leq$. By $\mathcal{T}$-LEARN, we add $C_3 \triangleq L_3^{\mathbb{B}} = \{\neg\ell_1, \neg\ell_3\}$ to the sequence of clauses in the Fixed-Alphabet DPLL($\mathcal{T}$) proof, and obtain C_1, C_2, C_3. Similarly, we can derive the clause $C_4 = \{\neg\ell_2, \neg\ell_3\}$ using $\mathcal{T}$-LEARN, and add it to the sequence. By propositional resolution RES on ℓ_1 using C_1 and C_3, we extend the sequence with the new clause $C_5 = C_1 \otimes_{\ell_1} C_3 = \{\ell_2, \neg\ell_3\}$. Similarly, we can derive $C_6 = C_4 \otimes_{\ell_2} C_5 = \{\neg\ell_3\}$ by RES. With a final application of RES, we conclude $C_7 = C_2 \otimes_{\ell_3} C_6 = \square$. Therefore, the sequence $C_1, C_2, C_3, \ldots, C_7$ constitutes a Fixed-Alphabet DPLL($\mathcal{T}$) proof of the fact that the input formula ϕ is unsatisfiable.

5.4 Non-interfering critical assignments

In this section, we give a new general theorem for establishing lower bounds on the number of $\mathcal{T}$-conflicts in Fixed-Alphabet DPLL($\mathcal{T}$) proofs of unsatisfiability (Section 5.3) for a fixed input formula ϕ. The theorem is based on the notion of sets of *non-interfering critical assignments* for ϕ. Informally, non-interfering critical assignments provide the means to measure the complexity of an SMT-based encoding of a problem instance independent from a particular implementation of an SMT solver based on the Fixed-Alphabet DPLL($\mathcal{T}$) framework.

To define non-interfering critical assignments, we recall the following standard notions. Recall that $\neg L$ denotes a $\mathcal{T}$-conflict (see Section 5.3). A $\neg L$-conflict is called *minimal* whenever every strict subset of $\neg L$ is $\mathcal{T}$-satisfiable. A *partial assignment* M is a set of literals in $\mathcal{L}_{\mathcal{X}}$ that does not contain both a variable and its negation. Partial assignments are interpreted as conjunctions, $\bigwedge_{\ell \in M} \ell$, and are always propositionally consistent. An *assignment* M is a partial assignment such that, for all propositional variables $v \in \mathcal{X}$, either $v \in M$ or $\neg v \in M$. Given a $\mathcal{T}$-formula ϕ, an assignment M is *critical* if it satisfies the initial propositional abstraction of ϕ (i.e., $M \models \bigwedge_{i=1}^{\alpha} C_i$) and there is exactly one minimal $\mathcal{T}$-conflict $\neg L$ such that $\neg L^{\mathbb{B}} \subseteq M$. We denote by Q a set of critical assignments for ϕ, all of which can be enumerated as $M_1, \ldots, M_{|Q|}$ and where $\neg L_i$ denotes the minimal $\mathcal{T}$-conflict for M_i. We say that Q is *non-interfering* whenever, for all distinct critical assignments $M_i \neq M_j$ in Q, $\neg L_i^{\mathbb{B}}$ is not a subset of M_j. In other words, no two critical assignments in Q contain the same $\mathcal{T}$-conflict.

Example 5.4.1. The assignments M_1 and M_2 in Example 5.3.1 are critical: the $\mathcal{T}$-conflict $\neg L_1 = \{x < y, y < x\}$ in M_1 is minimal and unique in the sense that there is no other subset of M_1 that leads to a $\mathcal{T}$-conflict. Moreover, M_2 contains exactly one minimal $\mathcal{T}$-conflict, namely $\neg L_2 = \{x = y, y < x\}$. The propositional abstraction

of both $\mathcal{T}$-conflicts are the clauses $\neg L_1^{\mathbb{B}} = \{\ell_1, \ell_3\}$ and $\neg L_2^{\mathbb{B}} = \{\ell_2, \ell_3\}$, respectively. Since neither $L_1^{\mathbb{B}}$ nor $L_2^{\mathbb{B}}$ is a subset of each other, $Q = \{M_1, M_2\}$ is a non-interfering set of critical assignments for the input formula ϕ in Example 5.3.1.

Lemma 5.4.2. *Let M be a critical assignment for ϕ with the minimal $\mathcal{T}$-conflict $\neg L$, and Π be a Fixed-Alphabet DPLL($\mathcal{T}$) proof that ϕ is unsatisfiable. There is a $\mathcal{T}$-LEARN application yielding $C_k \in \Pi$ such that $\neg L^{\mathbb{B}} \subseteq \neg C_k \subseteq M$.*

Proof. The assignment M does not satisfy the last clause $C_\beta = \square$ in Π. Therefore, there is some first clause C_k in Π that M does not satisfy, i.e. $M \not\models C_k$. By definition, the clause C_k cannot be an input clause as $M \models C_i$ for $1 \leq i \leq \alpha$. Additionally, C_k cannot be the result of RES: since C_k is the first unsatisfied clause, all $M \models C_i$ for $i < k$, and resolving C_i and $C_{i'}$ for $i' < i$ results in a clause satisfied by M. Thus C_k must be the result of a $\mathcal{T}$-LEARN application and $M \not\models C_k$. Since M is an assignment that does not satisfy C_k, M must contain the negation of all literals in C_k. Equivalently, $\neg C_k \subseteq M$. Let T be the $\mathcal{T}$-lemma corresponding to C_k: $C_k = T^{\mathbb{B}}$. As $\neg L^{\mathbb{B}}$ is the unique minimal subset of M that maps to a minimal theory conflict, $L \subseteq T$. Therefore, $\neg L^{\mathbb{B}} \subseteq \neg C_k \subseteq M$. □

Intuitively, Lemma 5.4.2 states that, for each critical assignment M, every conceivable Fixed-Alphabet DPLL($\mathcal{T}$) proof of unsatisfiability must contain a clause, derived by $\mathcal{T}$-LEARN, that rules out M as a model of ϕ in the theory $\mathcal{T}$.

Theorem 5.4.3. *Let ϕ be an unsatisfiable $\mathcal{T}$-formula, and let Q be a non-interfering set of critical assignments for ϕ. Then all Fixed-Alphabet DPLL($\mathcal{T}$) proofs that ϕ is unsatisfiable contain at least $|Q|$ applications of $\mathcal{T}$-LEARN.*

Proof. Let Π be any Fixed-Alphabet DPLL($\mathcal{T}$) proof. We will show that there exists a surjective partial map from $\mathcal{T}$-lemmas in Π onto critical assignments in Q that contain the same $\mathcal{T}$-inconsistency, proving the lower bound. We examine the set of partial maps $\mathbf{F}$ over $(\alpha, \beta]$ indices such that $\mathbf{F}(k) = j$ only if $L_j^{\mathbb{B}} \subseteq C_k$ and C_k is the result of a $\mathcal{T}$-LEARN application. Let $\mathbf{F}^*$ be a partial function that maps onto the maximal number of distinct $M \in Q$ among all such maps $\mathbf{F}$. If $\mathbf{F}^*$ maps onto all elements in Q, there are at least $|Q|$ applications $\mathcal{T}$-LEARN in Π. If $|Q| = 0$, the property trivially holds on Π.

For the remainder of this proof, assume that $|Q| \geq 1$. Suppose for contradiction that $\mathbf{F}^*$ is not surjective. We can then select some critical assignment $M_j \in Q$ such that for all $k \in (\alpha, \beta]$ either k is not in the domain of $\mathbf{F}^*$ (which is the case when C_k was not derived by $\mathcal{T}$-LEARN) or $\mathbf{F}^*(k) \neq j$.

By Lemma 5.4.2, there exists a $\mathcal{T}$-LEARN application that results in a clause $C_k \in \Pi$ such that $\neg L_j^{\mathbb{B}} \subseteq \neg C_k \subseteq M_j$. As $L_j^{\mathbb{B}} \subseteq C_k$, we know that it is possible for $\mathbf{F}^*$ to map C_k to some $M_m \in Q$. As $\mathbf{F}^*$ is maximal and there is no conflict mapped to M_j, $\mathbf{F}^*(k) = m$ for some $m \neq j$. By the construction of $\mathbf{F}^*$, $L_m^{\mathbb{B}} \subseteq C_k$. Recall that $\neg C_k \subseteq M_j$. Thus $\neg L_m^{\mathbb{B}} \subseteq \neg C_k \subseteq M_j$. As M_j contains both $\neg L_j^{\mathbb{B}}$ and $\neg L_m^{\mathbb{B}}$ for some distinct M_m in Q, this contradicts the assumption that Q is non-interfering.

We conclude that $\mathbf{F}^*$ maps some clause that is the result of $\mathcal{T}$-LEARN in Π onto each $M \in Q$. Therefore, Π contains at least $|Q|$ applications of $\mathcal{T}$-LEARN. □

Example 5.4.4. By Theorem 5.4.3 and the non-interfering critical assignments constructed in Example 5.4.1, we conclude that any Fixed-Alphabet DPLL($\mathcal{T}$) proof of the fact that the input formula ϕ in Example 5.3.1 is unsatisfiable requires at least two applications of the $\mathcal{T}$-LEARN rule.

There are many instances in the literature of *diamond benchmarks* for which exponential lower bounds on the number of $\mathcal{T}$-conflicts have been given [SSB02, BDdM08, MKS09, AM13, HAMM14]. Theorem 5.4.3 can be seen as a generalization of the lower bound arguments for such diamond benchmarks. The next section is devoted to a novel application of Theorem 5.4.3.

5.5 Lower bounds for quadratic and cubic partial-order encodings

We show that the challenge problem from Section 5.2 requires DPLL($\mathcal{T}$) to enumerate at least $N!$ theory conflicts before it finds a proof of unsatisfiability, for both of the $\mathcal{E}^3$ or $\mathcal{E}^2$ encoding where N is the number of threads. By Theorem 5.4.3, it suffices to find a set Q that contains $N!$ non-interfering critical assignments.

The intuition behind constructing this set of non-interfering critical assignments is as follows. We interleave the threads $\mathtt{T}_1, \ldots \mathtt{T}_N$ as if each thread were atomic. For example, for $\mathtt{T}_1, \mathtt{T}_2$ and $\mathtt{T}_3$ we get the following permutations:

$$\mathtt{T}_1; \mathtt{T}_2; \mathtt{T}_3; \mathtt{T}_0 \qquad (\pi_1)$$

$$\mathtt{T}_2; \mathtt{T}_1; \mathtt{T}_3; \mathtt{T}_0 \qquad (\pi_2)$$

$$\mathtt{T}_2; \mathtt{T}_3; \mathtt{T}_1; \mathtt{T}_0 \qquad (\pi_3)$$

$$\ldots \qquad (\pi_k)$$

Note that we do not need to interleave the instructions of individual threads themselves because we merely aim at a lower bound proof complexity result. It suffices to show that each of these restricted interleavings is satisfiable in the combined theory of clock and selection constraints, $\mathcal{T}_C + \mathcal{T}_S$, but leads to a unique minimal $\mathcal{T}_V$-conflict in the theory of value constraints:

$$\mathsf{rv}_1 = 0 \wedge \mathsf{rv}_2 = \mathsf{rv}_1 + 1 \wedge \mathsf{rv}_3 = \mathsf{rv}_2 + 1 \wedge \mathsf{rv}_{\text{assert}} = \mathsf{rv}_3 \wedge \mathsf{rv}_{\text{assert}} > N \qquad (\pi_1)$$

$$\mathsf{rv}_2 = 0 \wedge \mathsf{rv}_1 = \mathsf{rv}_2 + 1 \wedge \mathsf{rv}_3 = \mathsf{rv}_1 + 1 \wedge \mathsf{rv}_{\text{assert}} = \mathsf{rv}_3 \wedge \mathsf{rv}_{\text{assert}} > N \qquad (\pi_2)$$

$$\mathsf{rv}_2 = 0 \wedge \mathsf{rv}_3 = \mathsf{rv}_2 + 1 \wedge \mathsf{rv}_1 = \mathsf{rv}_3 + 1 \wedge \mathsf{rv}_{\text{assert}} = \mathsf{rv}_3 \wedge \mathsf{rv}_{\text{assert}} > N \qquad (\pi_3)$$

$$\ldots \qquad (\pi_k)$$

We formalize this as follows. Let S_N be the set of all permutations over $[1, N]$. Consider the following sequence of events that can be constructed from the permutation function π in S_N:

$$\sigma(\pi) : w_{init}, r_{\pi(1)}, w_{\pi(1)}, r_{\pi(2)}, w_{\pi(2)}, \ldots, r_{\pi(N)}, w_{\pi(N)}, r_{assert}.$$

The run of $\sigma(\pi)$ corresponds to satisfying the following clock and selection constraints:

- $\mathsf{c}_{w_{init}} \prec \mathsf{c}_{r_{\pi(1)}} \prec \mathsf{c}_{w_{\pi(1)}} \prec \cdots \prec \mathsf{c}_{r_{assert}}$,
- $\mathsf{s}_{w_{init}} = \mathsf{s}_{r_{\pi(1)}}$,
- $\bigwedge_{i=1\ldots N-1} \mathsf{s}_{w_{\pi(i)}} = \mathsf{s}_{r_{\pi(i+1)}}$,
- $\mathsf{s}_{w_{\pi(N)}} = \mathsf{s}_{r_{assert}}$

with distinct values for all s_w variables. A first-order variable assignment ν_π can be constructed to satisfy the above constraints. (An explicit construction of ν_π and proofs for Lemma 5.5.1 and Theorem 5.5.3 are given in Appendix G.) For each $\mathcal{T}_C$ or $\mathcal{T}_S$ literal ℓ, we include $\ell^{\mathbb{B}}$ in an assignment M_π if ℓ holds under ν_π. Consider the following $\mathcal{T}_V$-conflict:

$$\neg L_\pi = \{\mathsf{rv}_{r_{\pi(1)}} = 0\} \cup \{\mathsf{rv}_{r_{\pi(i)}} + 1 = \mathsf{rv}_{r_{\pi(i+1)}} \mid i = 1 \ldots N-1\} \cup \{\mathsf{rv}_{r_{\pi(N)}} + 1 = \mathsf{rv}_{r_{assert}}\} \cup \{\mathsf{rv}_{r_{assert}} > N\}.$$

Note that each $\ell \in \neg L_\pi$ is unit-propagated by the $\mathcal{T}_C$-literals and $\mathcal{T}_S$-literals already in M_π on the propositional abstraction of ϕ^3. We add $\neg L_\pi^{\mathbb{B}}$ to M_π. The remaining equality $\mathcal{T}_V$-atoms in ϕ^3 are added negatively. Now M_π satisfies the propositional abstraction of ϕ^3.

Lemma 5.5.1. *The assignment M_π is a critical assignment for ϕ^3 with the theory conflict $\neg L_\pi$.*

Theorem 5.5.2. *All Fixed-Alphabet DPLL($\mathcal{T}$) proofs for ϕ^3 contain at least $N!$ applications of $\mathcal{T}$-LEARN.*

Proof. Let $Q = \{M_\pi \mid \pi \in S_N\}$. For each pair of distinct π and π' in S_N, there is some adjacent pair of events with a different order in $\sigma(\pi)$ and $\sigma(\pi')$. Select k so that $\langle r_{\pi(k)}, r_{\pi(k+1)}\rangle \neq \langle r_{\pi'(k)}, r_{\pi'(k+1)}\rangle$. The literal $(\mathsf{rv}_{r_{\pi(k)}} + 1 = \mathsf{rv}_{r_{\pi(k+1)}})^{\mathbb{B}}$ is in $\neg L_\pi$ and is not in $M_{\pi'}$. Thus $\neg L_\pi^{\mathbb{B}}$ is not a subset of $M_{\pi'}$, and Q is non-interfering. The conclusion follows directly from Theorem 5.4.3. □

Theorem 5.5.3. *Let ϕ^2 be the $\mathcal{E}^2$ encoding of the challenge problem. All Fixed-Alphabet DPLL($\mathcal{T}$) proofs that ϕ^2 are unsatisfiable contain at least $N!$ application of $\mathcal{T}$-LEARN.*

An important difference between the diamond benchmarks and this problem is that for diamonds it is reasonable to describe all minimal $\mathcal{T}$-conflicts as they each also correspond to critical models. For the `fkp` problem, the encoding is more complex, and there are other classes of $\mathcal{T}$-conflicts. The set Q identifies those $\mathcal{T}$-lemmas that *must* appear during solving.

5.6 Experiments

In this section, we give experimental results that confirm the lower bounds on the DPLL($\mathcal{T}$) proofs for the cubic-size ($\mathcal{E}^3$) and quadratic-size ($\mathcal{E}^2$) encodings of the challenge problem (Section 5.2). For this purpose, our experiments are carried out along three dimensions: we use four SMT solvers (Boolector [BB09], CVC4 [BCD+11], Yices2 [Dut14], and Z3 [dMB08]), and we evaluate both $\mathcal{E}^3$ and $\mathcal{E}^2$ with respect to four different SMT-LIB theory combinations. We are using multiple SMT solvers to assess the generality of our theory since it aims to describe a range of mainstream SMT solvers. All of our SMT-LIB benchmarks are publicly available online.[2]

We performed all experiments on a 64-bit machine running Linux 3.16 with two Intel Xeon 2.5 GHz cores and 4 GB of memory. The timeout for each individual benchmark is set to 1 hour. Recall that $\mathcal{E}^3$ and $\mathcal{E}^2$ are parameterized by three theories, $\mathcal{T}_C$, $\mathcal{T}_S$ and $\mathcal{T}_V$. We experiment with the theory of reals $\mathcal{T}_{\mathbb{R}}$, the theory of integers $\mathcal{T}_{\mathbb{Z}}$,

[2] https://github.com/ahorn/benchmarks/commit/d49ce820ce00831583a249d3ff148afac5

	$\mathcal{E}^3$								$\mathcal{E}^2$							
N	$\wedge$	$\vee$	$\Rightarrow$	$=$	$\neq$	$<$	$\leq$	$+$	$\wedge$	$\vee$	$\Rightarrow$	$=$	$\neq$	$<$	$\leq$	$+$
3	199	18	73	54	2	88	0	3	135	18	41	54	2	43	36	3
4	344	27	135	81	2	122	0	4	194	27	60	81	2	52	54	4
5	553	38	227	114	2	162	0	5	265	38	83	114	2	61	76	5
6	838	51	355	153	2	208	0	6	348	51	110	153	2	70	102	6
7	1211	66	525	198	2	260	0	7	443	66	141	198	2	79	132	7
8	1684	83	743	249	2	318	0	8	550	83	176	249	2	88	166	8
9	2269	102	1015	306	2	382	0	9	669	102	215	306	2	97	204	9

Table 5.1: Number of SMT-LIB terms for the concurrency challenge problem.

and the theory of bit-vectors $\mathcal{T}_{\mathbb{BV}}$. In our experiments, we instantiate $\langle \mathcal{T}_C, \mathcal{T}_S, \mathcal{T}_V \rangle$ to the following four configurations such that $\mathcal{T}_C = \mathcal{T}_S$:

(1) "real-clocks-int-val": $\langle \mathcal{T}_{\mathbb{R}}, \mathcal{T}_{\mathbb{R}}, \mathcal{T}_{\mathbb{Z}} \rangle$, (3) "bv-clocks-int-val": $\langle \mathcal{T}_{\mathbb{BV}}, \mathcal{T}_{\mathbb{BV}}, \mathcal{T}_{\mathbb{Z}} \rangle$, and
(2) "real-clocks-bv-val": $\langle \mathcal{T}_{\mathbb{R}}, \mathcal{T}_{\mathbb{R}}, \mathcal{T}_{\mathbb{BV}} \rangle$, (4) "bv-clocks-bv-val": $\langle \mathcal{T}_{\mathbb{BV}}, \mathcal{T}_{\mathbb{BV}}, \mathcal{T}_{\mathbb{BV}} \rangle$.

CVC4 and Z3 were run on all benchmarks. Boolector is only used on the fourth configuration, i.e. purely $\mathcal{T}_{\mathbb{BV}}$ benchmarks. Yices was run on the "real-clocks-int-val" and "bv-clocks-bv-val" configurations. We further distinguish between the SMT-LIB benchmarks by labelling them with $\mathcal{E}^3$ or $\mathcal{E}^2$. For example, 'real-clocks-bv-val-$\mathcal{E}^3$' identifies benchmarks generated with the cubic encoding in which $\mathcal{T}_C$, $\mathcal{T}_S$ and $\mathcal{T}_V$ are respectively instantiated as $\mathcal{T}_{\mathbb{R}}$, $\mathcal{T}_{\mathbb{R}}$, and $\mathcal{T}_{\mathbb{BV}}$.

For all the "*-bv-val" benchmarks (except CVC4 for "real-clocks-bv-val"), the solvers are essentially encoding the problem in propositional logic and using a SAT solver.[3] The process of encoding into propositional logic is known as *bit-blasting* and enables the solver to learn clauses not necessarily expressible in the original alphabet of the input atoms. We therefore call these solver and configuration pairs *bit-blasted combinations*. All other solver and configuration pairs are called *DPLL($\mathcal{T}$) combinations*. The DPLL($\mathcal{T}$) combinations are the "*-int-val" configurations, and the run of CVC4 on "real-clocks-bv-val". As noted before, in this configuration CVC4 does not eagerly reduce $\mathcal{T}_{\mathbb{BV}}$ to SAT. DPLL($\mathcal{T}$) combinations use Fixed-Alphabet DPLL($\mathcal{T}$) proofs, whereas bit-blasted combinations generally do not. We still include the bit-blasted combinations because they allow us to evaluate the difficulty of the problem for SAT solvers.

Given an instantiation of $\langle \mathcal{T}_C, \mathcal{T}_S, \mathcal{T}_V \rangle$, we separately encode the concurrency chal-

[3] CVC4 was run with the flag `--bitblast=eager` on "bv-clocks-bv-val" benchmarks [HBJ+14].

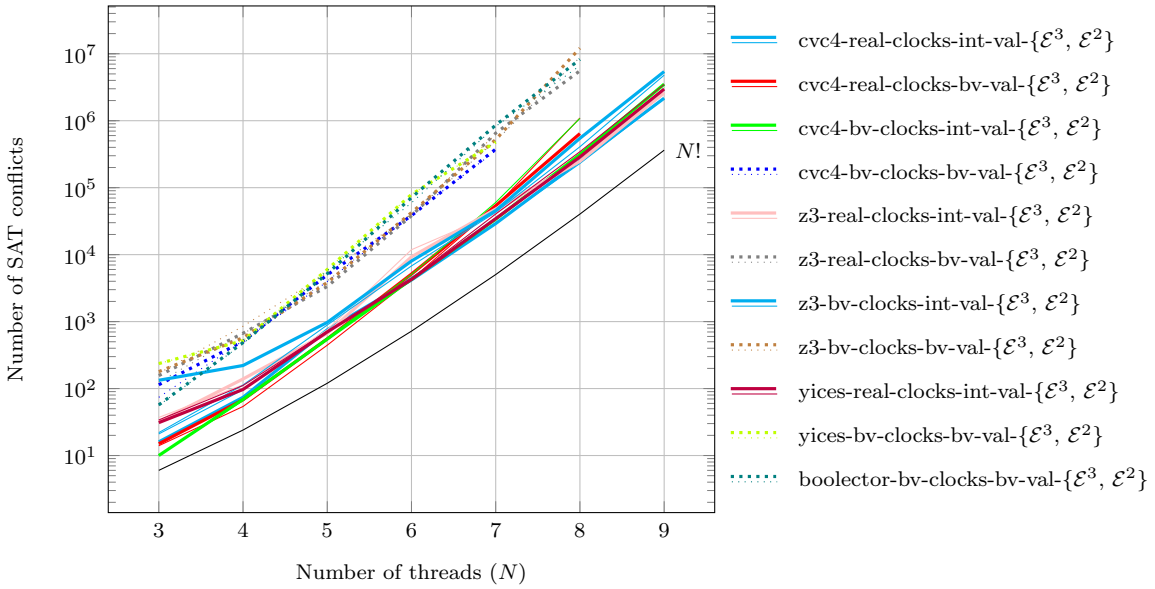


Figure 5.3: Experimental results for the concurrency challenge benchmark using four SMT solvers and four SMT-LIB theory combinations. The graph shows the factorial growth of the number of SAT conflicts in both the cubic-size and quadratic-size partial-order encoding as N increases.

lenge problem with $\mathcal{E}^3$ and $\mathcal{E}^2$ for all $N \in [3, 9]$. This experimental setup means that we have a total of $((N_{max}+1)-N_{min})\times 2\times 4 = 56$ unsatisfiable SMT-LIB benchmarks where $N_{min} = 3$ and $N_{max} = 9$. As a sanity check, we also temporarily modified the safety property such that it is violated. While these unsafe benchmark results are not reported in this chapter, we mention that in all supported cases, Boolector, CVC4, Yices, and Z3 find a counterexample that disproves the safety property for $N \leq 7$.

The size of each benchmark depends on N and whether we used $\mathcal{E}^3$ or $\mathcal{E}^2$. Table 5.1 gives the number of different kinds of symbolic expressions in the generated SMT-LIB formulas, namely the number of conjunctions ($\wedge$), disjunctions ($\vee$), implications ($\Rightarrow$), equalities ($=$), disequalities ($\neq$), strict ($<$) and non-strict ($\leq$) inequalities, and additions ($+$). Of course, the exact encoding of these symbolic expressions depends on the choice of SMT-LIB theory for $\mathcal{T}_C$ and $\mathcal{T}_V$. For example, $+$ is encoded either as integer arithmetic or fixed-size bit-vector addition. Irrespective of the choice of SMT-LIB theory, however, the formula generated by $\mathcal{E}^2$ is always smaller than $\mathcal{E}^3$'s. For example, for $N = 9$, the total number of symbolic expressions in $\mathcal{E}^3$ is 4085, whereas $\mathcal{E}^2$ yields only 1604 symbolic expressions.

The elapsed time and memory usage for the experiment is available in Table G.2 in Appendix G. Runs that exceeded the time limit of 1 hour are shown as 'TIMEOUT'. Figure 5.3 charts the number of conflicts reported by each solver during execution.

The x-axis corresponds to the number of threads, N. The y-axis corresponds to the number of conflicts generated by the solver and has a logarithmic scale. The legend for the chart groups together both the $\mathcal{E}^3$ (bold lines) and $\mathcal{E}^2$ (thin lines) for a solver and theory specification. These are further grouped into bit-blasted combinations (dotted lines) and DPLL($\mathcal{T}$) combinations (solid lines). We also plot $N!$ as a black line. The goal of the Figure 5.3 is to convey the overall trends instead of compare individual data points. The details of all our experimental results can be found in Table G.1, G.3 and G.4, summarized next.

We examine the number of SAT conflicts as it is a uniform and readily available statistic that is a lower bound on the number of proof steps taken by each solver. Across all combinations, the number of conflicts observed is above the $N!$ line. Thus the $N!$ theory conflict lower bound proofs given in Section 5.5 hold for the DPLL($\mathcal{T}$) combinations. Our theoretical lower bounds do not extend to the bit-blasted combinations. Nevertheless, our experiments show that the number of SAT conflicts is two orders of magnitude higher than $N!$ for bit-blasted combinations. We therefore conjecture that a similar $N!$ lower bound exists for Res proofs for the bit-blasted combinations.

We also examined CVC4's more detailed statistics on the DPLL($\mathcal{T}$) combinations. Doing so, we found that the factorial growth is due to the case split over the 'read-from' constraints in the theory combination $\mathcal{T}_S + \mathcal{T}_V$. In the case of CVC4, this would explain why both partial-order encodings suffer a significant performance penalty as N increases. We confirmed this by counting the theory conflicts in $\mathcal{T}_S + \mathcal{T}_V$ for "cvc4-real-clocks-bv-val-*" and "cvc4-bv-clocks-int-values-*" experiments. This showed that the number of fixed-size bit-vector and arithmetic conflicts blows up in the former and latter, respectively, but not vice versa. This therefore provides solver-specific evidence that the explosion of conflicts is not due to clock constraints but value constraints.

5.7 Interval theory lemmas

As we have seen, the crux of the concurrency challenge problem is that there is no Fixed-Alphabet DPLL($\mathcal{T}$) proof that is polynomial in size. Informally, the reason for this is that the alphabet in the input formula is too restricted. It is therefore natural to ask what a more expressive alphabet should look like in order for the DPLL($\mathcal{T}$) framework to be more effective at solving the concurrency challenge problem. Fortunately, the lower bound proofs (Section 5.5) and experiments (Section 5.6) suggest where to look for a solution: the theory of value constraints. Ultimately, the goal is

to reduce the number of $\mathcal{T}_V$-conflicts that are needed to prove unsatisfiability.

To make this concrete, we consider $\mathcal{T}_V$-lemmas in the form of unbounded intervals: $val(w) \leq k \vee k \leq val(w)$ for all write events w and integers $1 \leq k \leq 9$. We call these disjunctions *interval theory lemmas*. Their purpose is to introduce new $\mathcal{T}_V$-atoms that could be conceived to appear in possible short pen-and-paper proofs. The goal of our experiments is to measure how SMT solvers make use of the newly introduced $\mathcal{T}_V$-atoms in their own proofs. If $\mathcal{T}_V$ is a linear integer arithmetic theory, we show that most SMT solvers are several orders of magnitude faster after adding the interval theory lemmas. To show this, we consider the concurrency challenge problem of the most difficult level considered so far, namely $N = 9$. We first explain experiments with a simplified version of the concurrency challenge problem.

The simplified version of the concurrency challenge problem is designed for the quantifier-free linear integer arithmetic theory (`QF_LIA`), which we call *simplified `QF_LIA` encoding*.[4] Instead of using partial-orders as done in Section 5.2, our simplified `QF_LIA` encoding relies on integer constants to constrain $\mathcal{T}_V$-variables x_k for every $1 \leq k \leq N = 9$. The intuition is that x_k corresponds to the value at memory location x at step k. While this encoding is inherently restricted to interleaving semantics and a single memory location, it is now easy to see that $x_k \leq k$ holds for all $1 \leq k \leq 9$. Before adding the interval theory lemmas that we expect the SMT solver to use for inferring these upper bounds automatically, CVC4, MathSAT5, Yices and Z3 require over 4 hours to derive a DPLL($\mathcal{T}$) proof of unsatisfiability.[5] After adding the interval theory lemmas, all four SMT solvers find a proof of unsatisfiability in less than 1 minute, confirming our expectation. We repeated the experiment by changing the theory lemmas to be intervals over read events. In that case, the SMT solving is still several orders of magnitude faster but takes slightly longer, around 3 minutes.

Finally, we repeat the experiments from Section 5.6 using the $\mathcal{E}^3$ and $\mathcal{E}^2$ partial-order encodings when $\mathcal{T}_V = \mathcal{T}_{\mathbb{Z}}$ and $N = 9$ but this time with the extra interval theory lemmas. The experimental results are shown in Table G.5 and G.6. In the case of CVC4 using the quadratic-size partial-order encoding, the run-time is almost 4 hours despite the fact that CVC4 runs in less than 10 minutes when the interval theory lemmas are added to the cubic-size partial-order encoding. For Yices2, Z3 and MathSAT5, the problem with interval theory lemmas can be solved in less than 12 minutes, irrespective of the choice of $\mathcal{E}^3$ and $\mathcal{E}^2$ encoding, matching the speedups measured in the experiments with the simplified `QF_LIA` encoding (see previous paragraph).

[4] https://github.com/ahorn/benchmarks/commit/b65cbd2f3398e0dcaeb07a412b52d06fc4
[5] https://github.com/ahorn/benchmarks/commit/f0fd72ad211a2b94dba5a8b3b090a396e8

We found that the interval theory lemmas cannot speed up the SMT solving time when $\mathcal{T}_V = \mathcal{T}_{\mathbb{BV}}$. Note that this is expected for bit-blasted combinations, such as $\mathcal{T}_C = \mathcal{T}_V = \mathcal{T}_S = \mathcal{T}_{\mathbb{BV}}$. As a sanity check, we also explicitly disable simplifications in CVC4 to ensure that the interval theory lemmas are not pruned from the input formula. Despite this, the lazy bit-blasting combination using CVC4 appears unaffected by the interval theory lemmas, showing that our theory is not well suited to make accurate predictions about upper bounds (rather than lower bounds). In the next section, we consider the question whether interval theory lemmas can be learnt automatically with recently proposed extensions of DPLL($\mathcal{T}$), e.g. [DHK13, dMJ13, BDG+14].

5.8 Extensions of DPLL($\mathcal{T}$)

Encouraged by our results for $\mathcal{T}_V = \mathcal{T}_{\mathbb{Z}}$ in Section 5.7, we consider more general versions of DPLL($\mathcal{T}$) that impose fewer restriction on the alphabet of $\mathcal{T}$-atoms, e.g. [DHK13, dMJ13, BDG+14]. In particular, it is reasonable to postulate that the unbounded intervals in the interval theory lemmas from Section 5.7 can be represented in the interval domain used in MathSAT5's implementation of the *abstract conflict-driven clause learning* (ACDCL) algorithm [DHK13].

To experimentally evaluate ACDCL on this problem, we change the integer variables in the simplified `QF_LIA` encoding to 16-bit floats because MathSAT5 only supports ACDCL for floating-point arithmetic [BDG+14]. We purposefully avoid special floating-point values, such as infinity, by asserting each float variable to be in the range 0 to 1024. Our experiments confirm that MathSAT5 with ACDCL for the interval domain times out after 5 hours. Moreover, we run Yices2 with the MCSAT algorithm [dMJ13]; it also times out. This indicates that neither ACDCL nor MCSAT in their current form are an easy fix to the problem.

The research question therefore remains on how to automatically identify more useful $\mathcal{T}_V$-atoms, if possible, rather than requiring user guidance. Our experiments suggest avenues for future research into a restricted class of problems. In general, it is not clear whether the problem is decidable since the theory value of constraints (unlike clock constraints) can have an infinite number of ground atoms (see also [KS08, p. 246] for a more detailed discussion).

5.9 Concluding remarks

We have demonstrated a theoretical factorial lower bound for DPLL($\mathcal{T}$) proofs of a concurrency challenge problem using the cubic-size and quadratic-size partial-order encodings explained in Chapter 4. Our experiments confirm the theoretical lower bound for DPLL($\mathcal{T}$) proofs and show a strong relationship to the number of SAT conflicts in RES-proofs for bit-blasted fixed-size bit-vector encodings. Both the theoretical and empirical relationships hold over the cubic-size and quadratic-size encoding, which makes the results of our work particularly significant for state-of-the-art tools such as CBMC.

More generally, the kind of analysis we have undertaken throughout this chapter provides an important diagnostic practice in the development of SMT encodings. Previous research in this area has focused on clock constraints such as the 'write-serialization' and 'from-read' axioms [AMSS12, AKT13, HK15]. Our experiments, however, highlight the importance of case splits due to value constraints and the 'read-from' function, which so far has remained largely unexplored from a symbolic encoding perspective.

Our work therefore can direct future work into handling the value constraints for partial-order encodings and improving the performance of SMT solvers on such benchmarks by moving outside the confines of Fixed-Alphabet DPLL($\mathcal{T}$) proofs. In particular, the potential of expanding the alphabet on demand was demonstrated by manually adding interval theory lemmas that reduced the linear arithmetic theory solving time from several hours to a few minutes. We showed that these interval theory lemmas could not be automatically inferred by SMT solvers that implement a variant of ACDCL [DHK13] or MCSAT [dMJ13], highlighting opportunities for further research in this area.

Chapter 6

Partitioning of interval orders for faster linearizability checking of concurrent key/value data types

6.1 Introduction

Linearizability [HW90] is a well-established correctness criterion for concurrent data types and it corresponds to 'C' in the 'CAP theorem' [GL02]. This means that linearizability is also fundamental to studying distributed systems that are either consistent and available ('CA'), or consistent and partition tolerant ('CP').[1] The intuition behind linearizability is that every operation on a concurrent data type is guaranteed to take effect instantaneously at some point between its call and return.

The significance of linearizability for contemporary distributed key/value stores has been highlighted recently by the *Jepsen* project, an extensive case study into the correctness of distributed systems.[2] Interestingly, Jepsen found linearizability bugs in several distributed key/value stores despite the fact that they were designed based on formally verified distributed consensus protocols. A similar problem occurred for the pseudocode of a lock-free lazy linked-list set data structure [HHL+06], which had an unknown bug that went undetected in the subsequent machine-checked formal correctness proof [CGLM06] due to differences between the source code used in the proof and the pseudocode [BAM07]. This illustrates that there is often a gap between the design and the implementation of distributed systems and shared memory programs. This spurns on research into run-time verification techniques (in the form of so-called

[1] Depending on the reliability of the network [BK14], 'CA' distributed systems are less practical.
[2] https://aphyr.com/posts/316-call-me-maybe-etcd-and-consul

linearizability checkers) for finding linearizability bugs in concurrent systems.

We assume that the concurrent system under scrutiny is deadlock-free since there already exist good deadlock detection tools. The input to a linearizability checker consists of a specification that describes the pre- and post-conditions of the operations on the concurrent data type under scrutiny, and a certain partially ordered set of these operations, called a *history*. The purpose of the former is to formalize the expected behaviour of the operations on the concurrent data type, whereas the latter corresponds to a particular terminating run of a concurrent system in which these operations have been called (possibly from different threads of execution). Informally, the goal of a linearizability checker is to find out whether a given history is consistent with the specification of the concurrent data type.

Despite the restriction to single histories, linearizability checking is still in general an *NP*-complete problem [GK97].[3] This high computational complexity means that writing linearizability checkers that can be used in practice is difficult but arguably not entirely impossible as witnessed by the success of SAT solvers that can often solve reasonably quickly even large problem instances from industry. One of the keys to making this possible, however, is the right choice of data structures. In the case of linearizability checking, the crucial fact is that a history belongs a special class of partial-orders, called *interval orders* [BEEH15], that can be efficiently represented by an acyclic doubly linked-list [WG93]. This data structure is at the core of a recently devised backtracking algorithm [Low15] whose purpose is to explore a potentially huge search space.

This chapter presents a novel linearizability checker that efficiently prunes this search space by partitioning it into independent, faster to solve, subproblems. To achieve this, we propose *P-compositionality* (Definition 6.3.1), a *new partitioning scheme* of which Herlihy and Wing's locality principle [HW90] is an instance. Recall that locality says that a concurrent system Q is linearizable if and only if each concurrent object in Q is linearizable. The crux of P-compositionality is that it generalizes the idea behind the locality principle to operations on the same concurrent object. For example, the operations on a concurrent unordered set and map are linearizable if and only if the *restriction to each key* is linearizable. This is not a consequence of Herlihy and Wing's locality principle.

In this chapter, we study the pragmatics of P-compositionality through its implementation in a novel linearizability checker and experimental evaluation. Our

[3] In the case of a history of a read-write register, the linearizability problem becomes polynomial-time solvable when the number of processes is bounded [GK97].

implementation is based on Wing and Gong's algorithm (*WG algorithm*) [WG93] and a recent extension by Lowe [Low15]. We call Lowe's extension of Wing and Gong's algorithm the *WGL algorithm*. The idea behind the WGL algorithm is to prune states that are equivalent to an already seen state. Lowe's experiments show that the WGL algorithm can solve a significantly larger number of problem instances than the WG algorithm. We therefore use the more recent WGL algorithm as our starting point.

Our linearizability checker preserves three practical properties of the algorithms in the WG-family that we deem important. First, our tool is precise, i.e. it reports no false alarms. This is particularly significant for evaluating large code bases, as effectively shown by the Jepsen project. Second, our tool takes as input an *executable specification* of the data type to be checked. This significantly simplifies the task of expressing the expected behaviour of a data type because one merely writes code, i.e. no expertise in formal modeling is required. Finally, our tool can be easily integrated with a range of run-time monitors to generate a history from a run of a concurrent system. This is essential to make it a viable run-time verification technique.

We experimentally evaluate our linearizability checker using nine different implementations of concurrent sets, including Intel's TBB library, as exemplars of P-compositionality. Our experiments show that our linearizability checker is at least one order of magnitude faster and/or more space efficient than the WGL algorithm. Overall, the results of our work can therefore dramatically increase the number of runs that can be checked for linearizability bugs in a given time budget.

Contributions The main contributions of this chapter are as follows:

1. We present *P-compositionality*, which generalizes the idea behind Herlihy and Wing's locality principle to operations on the same concurrent data type.

2. We implement P-compositionality in a novel linearizability checker, which is based on Lowe's implementation [Low15]. We describe important design considerations that matter for the implementation of this decision procedure.

3. We experimentally evaluate our linearizability checker on over nine lock-free and/or wait-free implementations of concurrent sets, including Intel's TBB library. Our experiments show that our linearizability checker is one order of magnitude faster and/or more space efficient than the state-of-the-art algorithm.

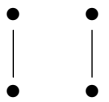
Organization The rest of this chapter is organized as follows. We first formalize the problem by recalling familiar concepts (Section 6.2). We then present P-compositionality (Section 6.3) on which our decision procedure (Section 6.4) is based. We implement and experimentally evaluate our decision procedure (Section 6.5). Finally, we discuss related work (Section 6.7) and conclude the chapter (Section 6.6).

6.2 Background

Before we make precise what it means for a history to represent a single terminating run of a concurrent system, we note that a history forms a certain type of partial-order [BEEH15], called interval order. We first recall the definition of an interval order.

Definition 6.2.1 (Interval order)**.** A partially ordered set $\langle E, \preceq \rangle$ is an **interval order** whenever, for all x, y, u, v in E, if $x \preceq y$ and $u \preceq v$, then $x \preceq v$ or $u \preceq y$.

Henceforth, we assume that interval orders are finite. The name interval order applies because their canonical representation is a family of closed intervals on the real line, say $\{I_x, I_y, \ldots\}$, where $x \preceq y$ holds whenever every point in the interval I_x is strictly less than every point in the interval I_y [Fis70]. Equivalently, a partial-order $\langle P, \leq \rangle$ is an interval order if and only if no restriction of $\langle P, \leq \rangle$ is isomorphic to the following Hasse diagram [Fis70, Rab78]:



A history H is defined to be two partial strings with same interval order but, in general, different labelling functions obj_H and op_H:

Definition 6.2.2 (History)**.** Fix Γ to be an alphabet. Define a **history** to be a tuple $H = \langle E_H, \preceq_H, obj_H, op_H \rangle$ where $\langle E_H, \preceq_H \rangle$ is an interval order, and $obj_H \colon E_H \to \Gamma$ as well as $op_H \colon E_H \to \Gamma$ are two labelling functions on the event set E_H. When no ambiguity arises, we simply write H for a history.

Intuitively, the purpose of a history H is to represent a particular run of a concurrent system. Using the implicitly associated labelling functions obj_H and op_H, a history H gives relevant information on all operations performed at run-time. That is to say, $\preceq_H$ corresponds to the happens-before relation. Given such a $\preceq_H$, recall that,

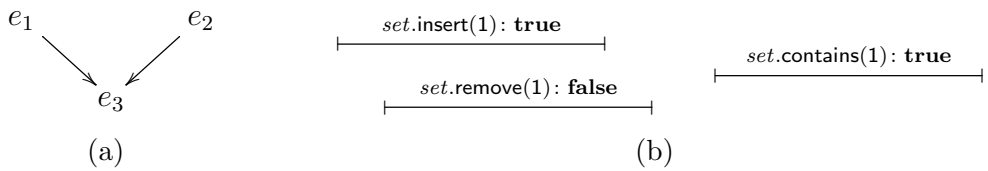


Figure 6.1: A history H_1 whose events satisfy the happens-before relation on the left (no labellings are shown). The history diagram on the right represents H_1 more compactly than the diagram on the left because it includes the labels of the three events.

for all events e and e' in E_H, we say that e *happens-before* e' whenever $e \preceq_H e'$, and both events are said to *happen concurrently* whenever neither $e \preceq_H e'$ nor $e' \preceq_H e$.

Example 6.2.3. Consider a concurrent set with the usual operations: 'insert' adds an element to a set, whereas 'remove' does the opposite, and 'contains' checks membership. The return value indicates the success of the operation. For example, 'set.remove(1): **true**' denotes the operation that successfully removed '1' from the object 'set', whereas 'set.remove(1): **false**' denotes the operation that did not modify 'set' because '1' is already absent from it, i.e. 'set.contains(1): **false**'. Consider the history H_1 in Figure 6.1a. We define H_1 such that $obj_{H_1}(e_i) =$ 'set', for all $1 \leq i \leq 3$, and op_{H_1} satisfies the following:

- $op_{H_1}(e_1) =$ 'insert(1): **true**',
- $op_{H_1}(e_2) =$ 'remove(1): **false**',
- $op_{H_1}(e_3) =$ 'contains(1): **true**'.

The events in H_1 satisfy the following happens-before relation:

- $e_1 \preceq_{H_1} e_3$ and $e_2 \preceq_{H_1} e_3$, i.e. both e_1 and e_2 happen-before e_3;
- $e_1 \not\preceq_{H_1} e_2$ and $e_2 \not\preceq_{H_1} e_1$, i.e. e_1 and e_2 happen concurrently.

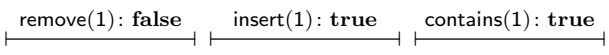
It can be shown that an interval order corresponds to the relative points in time at which an operation started and completed with respect to other operations [BEEH15]. This can be visualized using the familiar history diagrams [HW90], as illustrated next.

Example 6.2.4. The history diagram in Figure 6.1b visualizes the same history H_1 as defined in Example 6.2.3.

Henceforth, we draw diagrams as in Fig. 6.1b. Linearizability is ultimately defined in terms of sequential histories, in the following sense:

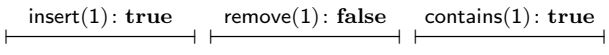
Definition 6.2.5 (Sequential history)**.** A history H is called **sequential** whenever $\preceq_H$ is a total order.

Example 6.2.6. The following history H_2 is sequential:



(H_2)

And so is H_3 that we get when we swap the first two operations in H_2 (although the resulting sequence of operations is not what we would expect from a sequential set, as discussed next):



(H_3)

H_3 in Example 6.2.6 illustrates that a history can be sequential even though it may not satisfy the expected sequential behaviour of the data type. This is addressed by the following definition:

Definition 6.2.7 (Specification)**.** A **specification**, denoted by ϕ (possibly with a subscript), is a unary predicate on sequential histories.

The happens-before relation $\preceq_H$ of a history H determines which of its events may be reordered in order to satisfy a specification.

Example 6.2.8. Define ϕ_{set} to be the specification of a sequential finite set. This means that, given a sequential history S according to Definition 6.2.5, the predicate $\phi_{set}(S)$ holds if and only if the input and output of 'insert', 'remove' and 'contains' in S are consistent with the operations on a set. For example, $\phi_{set}(H_2) = \textbf{true}$, whereas $\phi_{set}(H_3) = \textbf{false}$ for the histories from Example 6.2.6.

Remark 6.2.9. *In the upcoming decision procedure (Section 6.4), every ϕ is an* executable specification*. Informally, this is achieved by 'replaying' all operations in a sequential history S in the order in which they appear in S. If in any step the output deviates from the expected result, the executable specification returns false; otherwise, if it reaches the end of S, it returns true.*

Finally, the decision problem rests on the next definition:

Definition 6.2.10 (Linearizability)**.** A **ϕ-sequential history** is a sequential history H that satisfies $\phi(H)$. A history H is **linearizable with respect to** ϕ whenever there exists a ϕ-sequential history S such that the following holds:

L1 $E_H = E_S$, $obj_H = obj_S$ and $op_H = op_S$;

L2 $\preceq_H \subseteq \preceq_S$.

In other words, we can think of $\preceq_S$ as a topological sort of $\preceq_H$, i.e. $\preceq_S$ is a total order that preserves the interval order $\preceq_H$. For this definition to be adequate we recall our stated assumption (Section 6.1) that the concurrent system be deadlock-free [Low15]. Condition **L1** means that H' and S are identical if we disregard the order in which calls and returns occur in both sequences. Condition **L2** says that the happens-before relation between calls in H must be preserved in S.

Example 6.2.11. Recall Example 6.2.8. Then H_1 in Fig. 6.1b is linearizable with respect to ϕ_{set} because H_2 is a witness for a ϕ_{set}-sequential history that respects the happens-before relation $\preceq_{H_1}$ explained in Example 6.2.3. In particular, $e_1 \preceq_{H_1} e_3$ and $e_2 \preceq_{H_1} e_3$ cannot be reordered.

6.3 P-compositionality

In this section, we introduce P-compositionality. We illustrate our new partitioning scheme in Examples 6.3.3–3.5.

Definition 6.3.1 (P-compositionality)**.** Let P be a function that maps a history H to a non-trivial partition of H, i.e. P satisfies $P(H) \neq \{H\}$. A specification ϕ is called **P-compositional** whenever any history H is linearizable with respect to ϕ if and only if, for every history $H' \in P(H)$, H' is linearizable with respect to ϕ. When this equivalence holds we speak of **P-compositionality**.

In the following examples, we assume that the partitions are non-trivial. In fact, the first example illustrates that the locality principle [HW90] is an instance of P-compositionality.

Example 6.3.2. Denote with Obj the set of objects. Let ϕ be a specification for all objects in Obj. Let P_{Obj} be the function that maps every history H to the set of histories $\mathcal{H}$ where each sub-history $H' \in \mathcal{H}$ is the restriction of H to an object in Obj. Then $P_{Obj}(H)$ is a partition of H. By the locality principle [HW90], a history H is linearizable with respect to ϕ if and only if, for all $H_{obj} \in P_{Obj}(H)$, H_{obj} is linearizable with respect to ϕ. Therefore ϕ is a P_{Obj}-compositional specification.

The remaining examples show that P-compositionality strictly generalizes the locality principle because P-compositionality can partition a history even if the implementation details or the constituent parts (i.e. objects) of a concurrent system are unknown. For example, there are at least eight different implementations of concurrent sets (Table 6.2), but we do not need to know the objects (e.g. registers, buckets) of which such implementations consist in order to partition one of their histories. This is in contrast to the locality principle, where such knowledge is required. Put differently, P-compositionality is all about the *interface* of a concurrent data type, whereas the locality principle hinges on the *implementation details* of such an interface.

Example 6.3.3. Reconsider ϕ_{set}, the specification of a set from Example 6.2.8, where all operations have the form $\mathsf{insert}(k)$, $\mathsf{remove}(k)$ and $\mathsf{contains}(k)$ for some k. Let P_{set} be the function that partitions every history H according to such a k argument. Since the 'insert', 'remove' and 'contains' operations on a single set object are linearizable if and only if the restriction to each k is linearizable, ϕ_{set} is a P_{set}-compositional specification of a set.

Similarly, there exists a P_{map}-compositional specification for concurrent unordered maps where every history is partitioned by each key k.

Example 6.3.4. Consider a concurrent array. As their sequential counterparts, a concurrent array can be only read or written at a particular array index. Let P_{array} be the function that partitions a history based on such array indexes. This gives a P_{array}-compositional specification of an array.

Example 6.3.5. Consider a concurrent stack where each pop and push operation also returns the height of the stack before it is modified. Among other things, the return value can be used to determine whether the operation has succeeded. For example, if $stack.\mathsf{pop}$ returns zero, we know the pop operation was unsuccessful (and the popped element is undefined) because the stack was empty at the time the operation was called. We can use the returned height to partition a history such that a concurrent stack is linearizable if and only if each partition is linearizable. This way we get a P_{stack}-compositional specification of a stack.

Intuitively, the reason why the previous specifications in Examples 6.3.3–3.5 are P-compositional is that all operations in one partition are, informally speaking, unaffected by all operations in every other partition. For example, the return value of $set.\mathsf{insert}(k)$ is unaffected by $set.\mathsf{insert}(k')$, $set.\mathsf{remove}(k')$ and $set.\mathsf{contains}(k')$ for $k \neq k'$. This clearly, however, has its limitations. For example, a 'size' operation

that returns the number of elements in a concurrent collection data type cannot be generally partitioned this way.

Note that all these examples have in common that their P-compositional specifications can be expressed as a conjunction of specifications that each partition a history. For example, $\phi_{set} = \bigwedge_{k \in K} \phi_{set(k)}$ where $\phi_{set(k)}$ for every k is a sequential specification that only concerns operations on k, e.g. $set.\mathsf{insert}(k)$.

Next, we show how to leverage the concept of P-compositionality to more efficiently find linearizability bugs.

6.4 Decision procedure

In this section, we explain our linearizability checking algorithm that decides whether a history is linearizable with respect to some P-compositional specification (Definition 6.3.1). The novelty of our decision procedure is Algorithm 9 that leverages P-compositionality in conjunction with the WGL algorithm. In the next section (Section 6.5), we experimentally evaluate the effectiveness of the resulting linearizability checker.

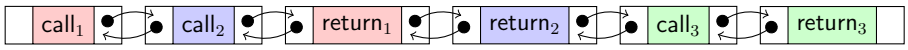
Of course, the data structures in a backtracking algorithm are important for efficiency reasons. Since we base our work on the WGL algorithm (recall Section 6.1), we use the following data structures to represent the input to the decision procedure:

1. The specification (Definition 6.2.7) is modelled by a persistent data structure, e.g. [Oka98]. Most standard data types in functional programming languages can be almost directly used this way. For instance, the specification of a set can be modelled through an immutable sequential set.

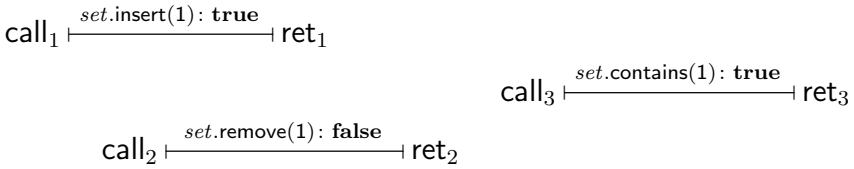
2. A history (Definition 6.2.2), in turn, is represented by an acyclic doubly-linked list of so-called **entries**. Consequently, each entry e has a $e.\mathsf{next}$ and $e.\mathsf{prev}$ field that point to the next and previous entry, respectively. In addition, each entry e has a match field, and we say that e is a **call entry** exactly if $e.\mathsf{match} \neq$ **null**; otherwise, e is called a **return entry**. Given a call entry e, $e.\mathsf{match}$ corresponds to the **matching return entry** of e. By [BEEH15], this linked-list data structure therefore aligns directly with the usual definition of history (Definition 6.2.2).

The entries in the doubly linked-list data structure come in pairs that are linked using the match field. Each pair of entries corresponds to the left and right bounds of a closed interval in the canonical representation of an interval order.

Example 6.4.1. The following doubly-linked list represents history H_1 shown in Figure 6.1b which we explained in Example 6.2.3:



The colouring and subscripts identify matching call and return entries. Each pair of these call/return nodes in the doubly-link corresponds to a single event in H_1. We visualize this correspondence by annotating history diagrams as follows:



The idea behind the WGL Algorithm 7 is threefold: it keeps track of provisionally linearized call entries in a stack; it uses the stack to backtrack if necessary, and caches already seen configurations. We briefly explain in turn the stack, and the idea behind using the history, stack and cache for backtracking purposes.

Denote the stack of call entries by calls. Given a history H, the height of calls is at most half the number of entries in H. Note that there is no rounding involved here because every call entry has a matching return entry. The height of the stack grows only if a call entry can be linearized (line 5). When the stack grows or shrinks, the history is modified (lines 13 and 23) by the LIFT and UNLIFT procedures (Algorithm 8). We remark that the workings of both procedures are illustrated by Example 6.4.2.

If no further call entries can be linearized but the stack is nonempty, the algorithm backtracks and tries the next possible call entry (lines 18–24). The backtracking points depend on the return value of *apply*(entry, s) and the cache. The former (line 3) models the specification ϕ: by Remark 6.2.9, it determines whether entry can be applied to the current state s of a persistent data type. The latter (lines 4–8) is an optimization due to Lowe [Low15] that prunes the search space by memoizing already seen configurations which are known to be non-linearizable. More accurately, each configuration is a pair that consists of a set of unique call entry identifiers and a state of the persistent data structure. The intuition behind pruning already seen configurations is that only one of two permutations of operations on a concurrent data type need to be considered if they lead to an identical state [Low15]. We remark that the total correctness of the WGL algorithm follows from Wing and Gong's total correctness argument [WG93].

Example 6.4.2. We illustrate the handling of entries in the history data structure. For this, consider the two histories in Fig. 6.2. In Fig. 6.2a, the entries satisfy the following: $\mathsf{call}_2.\mathsf{prev} = \mathsf{call}_1$, $\mathsf{call}_2.\mathsf{next} = \mathsf{call}_3$ and $\mathsf{call}_2.\mathsf{match} = \mathsf{ret}_2$ etc. Then LIFT(call_2) (Algorithm 8) produces the history shown in Fig. 6.2b. Note that both call_2 and ret_2 are still valid entry pointers whose fields remain unchanged. This explains how UNLIFT(call_2) reverts the change in constant-time.

Algorithm 9 gives our partitioning scheme. This is an iterative algorithm that, given an entry in a history H and positive integer n, partitions H starting from that entry into at most n separate sub-histories. The partitioning is controlled by the function $\mathit{partition} : \mathsf{Entry} \to \mathbb{N}$, which maps the set of call and return entries, Entry, to the natural numbers.

Example 6.4.3. Consider the history in Fig. 6.2b. For all entries e in this history, let $\mathit{partition}(e) = k$ where k is the integer argument of the operation. For example, $\mathit{partition}(\mathsf{call}_3) = \mathit{partition}(\mathsf{ret}_3) = 1$ because $\mathit{op}(\mathsf{call}_3)$ and $\mathit{op}(\mathsf{ret}_3)$ map to the operation that removes '1' from the same set, i.e. set.'$\mathsf{remove}(1) : \mathbf{false}$'. Then the function PARTITION(call_1) returns two disjoint sub-histories for the operations on '0' and '1', respectively:

$$\mathsf{call}_1 \overset{\mathit{set}.\mathsf{insert}(0):\,\mathbf{true}}{\longmapsto} \mathsf{ret}_1 \qquad \mathsf{call}_2 \overset{\mathit{set}.\mathsf{contains}(0):\,\mathbf{true}}{\longmapsto} \mathsf{ret}_2 \qquad \text{and} \qquad \mathsf{call}_3 \overset{\mathit{set}.\mathsf{remove}(1):\,\mathbf{false}}{\longmapsto} \mathsf{ret}_3.$$

Given a nonempty set of disjoint sub-histories returned by the PARTITION function (Algorithm 9), we invoke Algorithm 7 on each sub-history. It is not too difficult to implement sub-histories such that there is no sharing between them, and Algorithm 7 could be therefore run in parallel for each sub-history. This parallelization opportunity addresses a challenging problem that was identified independently by Lowe [Low15] and Kingsbury [Kin14].

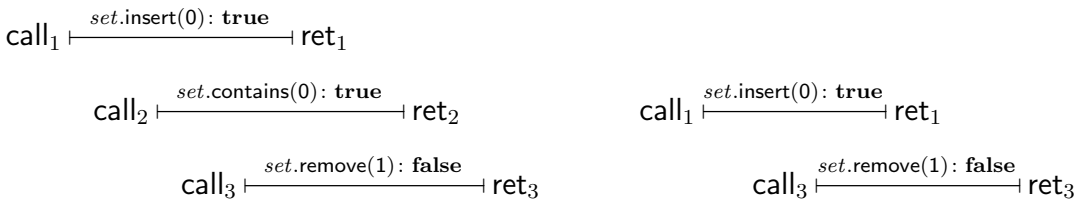


Figure 6.2: After calling LIFT(call_2) in history (a), we get the history in (b). UNLIFT(call_2) reverts this change in constant-time.

Algorithm 8 History modifications

```
1:  procedure LIFT(entry)
2:      entry.prev.next ← entry.next
3:      entry.next.prev ← entry.prev
4:      match ← entry.match
5:      match.prev.next ← match.next
6:      if match.next ≠ null then
7:          match.next.prev ← match.prev
8:
9:  procedure UNLIFT(entry)
10:     match ← entry.match
11:     match.prev.next ← match
12:     if match.next ≠ null then
13:         match.next.prev ← match
14:     entry.prev.next ← entry
15:     entry.next.prev ← entry
```

Algorithm 9 History partitioner

Require: n is a positive integer
Require: entries is an array of size n

```
1:  function PARTITION(entry, n)
2:      for 0 ≤ i < n do
3:          entries[i] ← null
4:      while entry ≠ null do
5:          i ← partition(entry) mod n
6:          if entries[i] ≠ null then
7:              entries[i].next ← entry
8:          next_entry ← entry.next
9:          entry.prev ← entries[i]
10:         entry.next ← null
11:         entries[i] ← entry
12:         entry ← next_entry
13:     return entries
```

Theorem 6.4.4. *Let ϕ be a P-compositional specification and H be a history. Denote by* head_entry *the entry that represents the beginning of H. Associate with each disjoint history H_k in partition $P(H)$ a unique number $0 \leq k < |P(H)| = n$. If, for all $H_k \in P(H)$ and $e \in H_k$, partition$(e) = k$, then H is linearizable with respect to ϕ if and only if Algorithm 7 returns true for every history in* PARTITION(head_entry, n).

We next experimentally quantify the benefits of the previous theorem.

6.5 Implementation and experiments

In this section, we discuss and experimentally evaluate our implementation of the decision procedure (Section 6.5). As an exemplar of P-compositionality, our experiments use Intel's TBB library and Lowe's implementations of concurrent sets.

6.5.1 Implementation

For experimental research purposes, the implementation details of our decision procedure matter. We particularly consider hashing and cache eviction options because these were not studied in previous implementations of the WG-based algorithms [WG93, Low15].

For experimental robustness, we implemented our linearizability checker in the C++11 programming language [ISO12] because it has built-in concurrency support

while allowing us to rule out interference from managed run-time environments (e.g. JVM) due to garbage collection. The choice of language, though, meant that we had to implement persistent data structures from scratch. In doing so, we focused on optimizing equality checks for our specific purposes. This way, we managed to avoid a known performance bottleneck in Lowe's implementation of the WGL algorithm [Low15] where the cost of equality checks had to be compensated with an additional union-find data structure. Another optimization in our implementation is a constant-time (instead of linear-time) hash function for bitsets where we exploit the fact that the bitwise XOR operator over fixed-size bit vectors forms an abelian group. This optimization turns out to be important when histories are longer than 8 K, see [Low15]. To see this, consider the computational steps for retrieving a configuration from the cache and updating it (line 8 in Algorithm 7). For example, a history of length 2^{16} means that each bitset in a configuration is at least 3 KiB, and so a constant-time hash function can make a measurable difference when the cache is frequently accessed. In fact, it is not uncommon for the cache to contain more than 27 K of such configurations. For this reason, we also implemented a *least recently used* (LRU) cache eviction feature that can optionally be enabled at compile-time. The effects of the LRU cache will be evaluated shortly.

Overall, our implementation and experimental setup is around 4 K lines of code, including several dozen unit tests. All the code and benchmarks are publicly available in our source code repository.[4]

6.5.2 TBB and concurrent set experiments

For the experimental evaluation of our partitioning scheme, we collected over 700 histories from nine different implementations of concurrent sets by Lowe [Low15] and the concurrent unordered set implementation in Intel's TBB library.[5] We remark that the authors in [VYY09] deem implementations of concurrent sets and maps more complex, and therefore higher risk candidates for bugs, than concurrent queues and stacks because the latter only operate on the endpoints of the underlying data structure. We performed all experiments on a 64-bit machine running GNU/Linux 3.17 with 12 Intel Xeon 2.4 GHz cores and 94 GB of main memory.

Each history is generated by running 4 concurrent threads that pseudo randomly invoke operations on a single shared concurrent set. The argument of each operation is a pseudo random uniformly distributed integer between 0 (inclusive) and 24 (ex-

[4] https://github.com/ahorn/linearizability-checker
[5] https://www.threadingbuildingblocks.org/

Benchmark	WGL			WGL+LRU			WGL+P		
	Time	Memory	Timeout	Time	Memory	Timeout	Time	Memory	Timeout
TBB	101 s	9792 MiB	0%	11 s	**670 MiB**	0%	**6 s**	672 MiB	0%
CRLSL	20 s	15738 MiB	0%	25 s	678 MiB	0%	**6 s**	**400 MiB**	0%
CRLFSL	14 s	15029 MiB	0%	18 s	678 MiB	0%	**5 s**	**401 MiB**	0%
FGL	16 s	14297 MiB	0%	81 s	678 MiB	0%	**5 s**	**401 MiB**	0%
LLL	23 s	16494 MiB	0%	94 s	678 MiB	0%	**6 s**	**401 MiB**	0%
LSL	20 s	15736 MiB	**0%**	25 s	678 MiB	14%	**6 s**	**401 MiB**	**0%**
LFLL	11 s	11847 MiB	0%	15 s	678 MiB	0%	**5 s**	**402 MiB**	0%
LFSL	14 s	14712 MiB	0%	18 s	678 MiB	0%	**5 s**	**401 MiB**	0%
LFSLF0	14 s	13125 MiB	0%	18 s	678 MiB	0%	**5 s**	**402 MiB**	0%
LFSLF1	< 1 s	404 MiB	0%	< 1 s	407 MiB	0%	< 1 s	402 MiB	0%
OPTIMIST	16 s	13818 MiB	**0%**	54 s	678 MiB	9%	**5 s**	**401 MiB**	**0%**

Table 6.1: Experimental results for three variants of the same linearizability checker. The results for the baseline are reported in the WGL column. The rows correspond to benchmarks drawn from Intel's TBB library and Lowe's implementations of concurrent sets (see Table 6.2 for mnemonics).

clusive). Each thread invokes 70 K such operations. Note that this is significantly more than in previous experiments where each process is limited to $2^{13} \approx 8$ K operations [Low15]. In total, since every call generates a pair of entries, every history H in our benchmarks has $4 \times 2 \times 70\,\text{K} = 560\,\text{K}$ entries. Next, we discuss the experimental results using Intel's TBB library and Lowe's implementations of concurrent sets in turn.

The experimental results are given in Table 6.1. Each of the three main columns corresponds to one variant of the same linearizability checker: 'WGL' is the baseline, 'WGL+LRU' is the WGL algorithm with LRU cache eviction enabled (Section 6.5.1), and 'WGL+P' is the WGL algorithm combined with our partitioning algorithm (Algorithm 9 in Section 6.4). Note that the LRU cache eviction is *disabled* in 'WPL+P'. We tried to use the WG algorithm [WG93] without the extension by Lowe [Low15] but WG times out on the majority of benchmarks. We therefore do not report the results on the WG algorithm and focus on WGL, WGL+LRU and WGL+P. The meaning of the sub-columns is as follows. The 'Time' and 'Memory' columns give the average of the elapsed time (seconds) and virtual memory usage (MiB), respectively. These averages exclude runs that we had to terminate after 1 hour. The percentage of such terminated runs is given in the 'Timeout' column. In each row, all variants are compared with respect to the same benchmark data. We therefore do not report confidence intervals.

The TBB benchmark corresponds to the first row in Table 6.1 and consists of a total of 100 histories. Table 6.1 clearly shows that the WGL+P algorithm is at least one order of magnitude faster compared to the baseline. We also see that enabling the LRU cache eviction decreases the memory footprint by at least one order of magnitude, approximately 10 GiB versus 700 MiB. In fact, the run-time performance of WGL+LRU is almost one order of magnitude faster than the baseline. The WGL+P algorithm is at least as fast and almost as space efficient as WGL+LRU. In the experiments with Lowe's implementations of concurrent sets (see next paragraph), we further investigate the effect of the LRU cache eviction feature and how it compares to the partitioning scheme.

We give Lowe's implementations of concurrent sets mnemonics (Table 6.2) that identify the remaining ten benchmarks in Table 6.1. Each of these ten benchmarks comprises between 50 and 100 histories with an average of 70 histories per benchmark. To avoid bias, we collected these using Lowe's tool. The significance of the experimental results in Table 6.1 is twofold. Firstly, they show that on average, WGL+P is three times faster than WGL, and WGL+P consumes one order of magnitude less space than WGL. Secondly, and more crucially, however, these experiments reveal that WGL+LRU is not as efficient as WGL+P, in neither time nor space. For example, for WGL+LRU the average elapsed time of the FGL and LLL benchmark is 81 s and 94 s, respectively, with an average memory usage of 678 MiB in both cases. By contrast, WGL+P achieves an average run-time of less than 7 s (and so WGL+P is one order of magnitude faster than WGL+LRU) and consumes even less memory on average (401 MiB) than WGL+LRU. The higher average run-time of WGL+LRU in the FGL benchmark is due to a single check that took several orders of magnitude longer (3068 s) than the remaining checks (20 s on average when the 3068 s outlier is excluded). In the LLL benchmark there are two such outliers (2201 s and 675 s, whereas the other checks average 27 s). The observed difference between WGL+LRU and WGL+P is even more pronounced in both the LSL and OPTIMIST benchmarks where the LRU cache eviction causes 14% and 9% of runs to timeout, whereas the WGL+P algorithm always runs to completion in less than a few seconds.

This experimentally confirms that the WGL+P is one order of magnitude faster as well as more space efficient than the baseline and WGL+P consumes even less space than our WGL+LRU implementation.

Benchmark name	Mnemonic	Benchmark name	Mnemonic
collision resistance lazy skip list	CRLSL	lock-free linked-list	LFLL
collision resistance lock-free skip list	CRLFSL	lock-free skip list	LFSL
fine-grained lock	FGL	lock-free skip list faulty (bad hash)	LFSLF0
lazy linked-list	LLL	lock-free skip list faulty (good hash)	LFSLF1
lazy skip list	LSL	optimistic lock	OPTIMIST

Table 6.2: Mnemonics for Lowe's implementation of concurrent sets [Low15]

6.5.3 Etcd experiments

It is natural to ask how the WG-family of algorithms compares to Jepsen's built-in linearizability checker, called Knossos.[6] For this comparison, we used Jepsen to collect histories from etcd, a distributed key/value store.[7] Since the specification of the etcd register-based client cannot be further partitioned, we only report on experimental results for Knossos, WGL and WGL+LRU, excluding WGL+P.

Out of 100 collected etcd histories using Jepsen, 80% are non-linearizable (here, this is expected because we allow read requests to bypass the consensus protocol), whereas the remaining 19% of etcd histories are linearizable and 1% is invalid due to an LXC system failure. On average, each etcd history has length 178, totaling around 8 K operations if all histories were combined. While this is significantly shorter than all the other histories in our experiments, the etcd histories are nevertheless interesting because the operations issued by the etcd client can timeout. This is due to the network link failures purposefully induced by Jepsen. On average, there are 13.6 timeouts in each history, approximately 15% of all operations in a history. We interpret a timed out call entry by placing its matching return entry at the end of the history because we do not know when the timed out operation has exactly completed. A timed out operation is therefore treated to happen concurrently with respect to every other subsequent operation. This makes the etcd histories different from those in the previous experiments (Section 6.5.2).

The results of our etcd experiments are as follows. Knossos times out (we terminated it after more than 12 hours) on benchmarks 7 and 99, and it runs out of memory on benchmarks 40, 57, 85 and 97. All of those benchmarks (except benchmark 7) are non-linearizable. In comparison, WGL+LRU times out (set to 1 hour) on benchmarks 2, 7, 57, 75, and 99, and WGL+LRU takes 1765 s and 2757 s on benchmarks 40 and 80, respectively. In contrast to WGL+LRU, Knossos completes

[6] https://github.com/aphyr/knossos
[7] https://github.com/coreos/etcd

benchmark 80 in approximately 2 min. In contrast to both WGL+LRU and Knossos, WGL checks all 100 benchmarks in around 7 s, averaging 0.07 s per benchmark. On these benchmarks, the memory usage of both WGL+LRU and WGL is negligible.

6.6 Concluding remarks

We have presented a precise, fully automatic run-time verification technique for finding linearizability bugs in implementations of concurrent data types that are expected to satisfy a P-compositional specification. Our experiments show that our partitioning scheme improves the WGL algorithm [WG93, Low15] by one order of magnitude, in both time and/or space. An additional strength of our technique is that it is applicable to any linearizability checker. It could be also applied to a purely symbolic analysis. This assumes that the specification is P-compositional. Since this is generally not always the case, however, it would be interesting to further generalize P-compositionality, perhaps with a less modular partitioning scheme that can make more assumptions about the underlying decision procedure.

6.7 Bibliographic notes

Linearizability is related to the concept of atomicity, including weaker forms such as k-atomicity [AAB05]. An important difference is that atomicity is typically merely defined in terms of a sequential specification of a read-write register, e.g. [WS05]. By contrast, linearizability is more general because it is not restricted to a particular concurrent data type. The theoretical limitations of automatically verifying linearizability are well understood. Of course, the problem is generally undecidable [BEEH13]. In fact, even checking finite-state implementation against atomic specifications, provided the number of program threads is bounded, is EXPSPACE [AMP00]. And the best known lower bound for this problem is PSPACE-hard. This explains the restrictions in this chapter and its focus on run-time verification instead.

The literature on machine-assisted techniques for checking linearizability is extensive and can be broadly divided into simulation-based methods (e.g. [CDG05, DSW11]), model checking (e.g. [VYY09, BDMT10, vRZ+10, LCL+13]), static analysis (e.g. [ARR+07, BLAM+08, Vaf09, Vaf10]) and fully automatic testing (e.g. [WG93, ALS+10, SBA+11, FLR11, PG12, PG13, GHL13, Low15]). The simulation-based methods have been used by experts to mechanically verify simple fine-grained and lock-free implementations. Model checking requires less expertise but is typically

limited to very small programs and a small number of threads due to the state explosion problem. By contrast, static analysis tools aim to prove correctness with respect to an unbounded number of threads. In general, these techniques are necessarily incomplete and require the user to supply linearization points and/or invariants. Vafeiadis [Vaf10] proposes a more automatic form of static analysis that works well on simpler concurrent data types such as stacks but reportedly not so well on data types that have more complicated invariants, including the CAS-based and lazy concurrent sets extensively studied in our experiments.

Our work is most closely related to linearizability testing techniques that are precise, fully automatic and necessarily incomplete, e.g. [WG93, ALS+10, SBA+11, FLR11, PG12, PG13, GHL13, Low15]. We focus our discussion on tools that do not require the user to specify linearizability points because it would trivialize the problem of checking the linearizability of individual histories, cf. [ETQ05]. The work in [ALS+10, GHL13] checks k-atomicity with a polynomial-time algorithm assuming that each write to a register assigns a distinct value. By contrast, we solve a more general problem of which k-atomicity is an instance. The tool in [SBA+11] analyzes code that uses concurrent collection data types such as maps. To make the analysis scale, the authors assume that the collection data types are linearizable, whereas our tool could be used to check such an assumption. A different tool [FLR11] requires programmers to annotate concurrent implementations with so-called state summary functions that act as a form of specification. Our approach is more modular because it strictly separates the concurrent implementation from its specification. By contrast, [PG12] works without the programmer having to provide a sequential specification. As a result, however, the tool can only find linearizability violations when an exception is thrown or a deadlock occurs. Subsequent work [PG13] circumvents this, in the context of object-oriented programs, by considering the special case of a superclass serving as an executable, possibly non-deterministic, specification for all its subclasses. The fact that the superclass can be non-deterministic may explain why even checks of two threads can take a significant amount of time (e.g. 108 min) despite the fact that each concurrent test considers only two possible linearizations [PG13]. By contrast, the WGL algorithm [WG93, Low15], on which our decision procedure is based (Section 6.4), is significantly faster but limited to deterministic specifications. Crucially, our experiments (Section 6.5) with P-compositional specifications show a significant improvement over the WGL algorithm.

Algorithm 7 WGL linearizability checker [Low15]

Require: Let H be a history.
Require: head_entry is such that head_entry.next points to the beginning of H.
Require: N is half of the total number of entries reachable from head_entry.
Require: linearized is a bitset (array of bits) with linearized$[k] = 0$ for all $0 \leq k < N$.
Require: For all entries e in H, $0 \leq entry\_id(e) < N$.
Require: For all entries e and e' in H, if $entry\_id(e) = entry\_id(e')$, then $e = e'$.
Require: cache is an empty set and calls is an empty stack.

1: **while** head_entry.next $\neq$ **null do**
2: **if** entry.match $\neq$ **null then** ▷ Is call entry?
3: $\langle$is_linearizable, s$'\rangle \leftarrow apply$(entry, s) ▷ Simulate entry's operation
4: cache$' \leftarrow$ cache ▷ Copy set
5: **if** is_linearizable **then**
6: linearized$' \leftarrow$ linearized ▷ Copy bitset
7: linearized$'[entry_id($entry$)] \leftarrow 1$ ▷ Insert $entry\_id$(entry) into bitset
8: cache $\leftarrow$ cache $\cup \{\langle$linearized$', $ s$'\rangle\}$ ▷ Update configuration cache
9: **if** cache$' \neq$ cache **then**
10: calls $\leftarrow push($calls$, \langle$entry, s$\rangle)$ ▷ Provisionally linearize call entry/state
11: s $\leftarrow$ s$'$ ▷ Update state of persistent data type
12: linearized$[entry_id($entry$)] \leftarrow 1$ ▷ Keep track of linearized entries
13: LIFT(entry) ▷ Provisionally remove the entry from the history
14: entry $\leftarrow$ head_entry.next ▷ Continue search in shortened history
15: **else** ▷ Cannot linearize call entry
16: entry $\leftarrow$ entry.next ▷ Continue search in unmodified history
17: **else** ▷ Handle "return entry"
18: **if** $is_empty($calls$)$ **then**
19: **return false** ▷ Cannot linearize entries in history
20: $\langle$entry, s$\rangle \leftarrow top($calls$)$ ▷ Revert to earlier state
21: linearized$[entry_id($entry$)] \leftarrow 0$
22: calls $\leftarrow pop($calls$)$
23: UNLIFT(entry) ▷ Undo provisional linearization
24: entry $\leftarrow$ entry.next
25: **return true**

Chapter 7

Conclusions

Chapter 2 constructed a simple, yet relevant and realistic, concurrency benchmark by extracting the OpenCores Ethernet MAC hardware model from the QEMU virtual machine. Our experiments revealed that CBMC with the state-of-the-art partial-order encoding [AKT13] dominated the total run-time rather than the SAT solver.

This rather surprising gap motivated the quadratic-size partial-order encoding in Chapter 4 (see Figure 4.9). The quadratic-size encoding can be constructed in an asymptotically smaller number of steps than the cubic-size encoding. This achievement was due to studying the partial-order constraints $\mathcal{T}_C$ of the cubic-size encoding.

Even though the asymptotic reduction in construction steps of the quadratic-size encoding was clearly reflected in our experiments, it came at the cost of new 'supremum' variables, one for every read event. This meant that the size of the CNF formula increased, an example of the kind of trade-offs faced in designing partial-order encodings. For our experimental comparison of both encodings, we purposefully opted to implement the new encoding in CBMC. This decision proved important for three main reasons. First, we were able to reuse the implementation of the state-of-the-art, thereby preserving its performance characteristics. Second, since CBMC is open-source, we were able to implement our encoding in the same framework as the cubic-size encoding, making our experimental comparison more robust and reliable. Finally, the maturity of CBMC made it possible to go beyond toy examples and run experiments on a range of larger shared memory programs written in C, such as the OpenCores Ethernet MAC benchmark from Chapter 2. We carefully designed our experiments along multiple dimensions such as fixing the ratio of reads and writes, using different SAT solvers, and adjusting the search heuristics in SAT solvers through the use of non-decision variables. This convincingly showed that MiniSAT performed significantly better on the quadratic-size encoding than Glucose, whereas Glucose per-

formed significantly better on the cubic-size encoding than MiniSAT (see Table 4.2). Since both SAT solvers share the same code base, this provides opportunities for future research into their search heuristics.

Still, irrespective of the choice of SAT solver, we found that on sufficiently large problems the quadratic-size encoding successfully increases the percentage of the SAT solving time from less than 40% to nearly 100%. Our encoding therefore achieves to close the gap we had earlier uncovered in Chapter 2. This achievement is important because it provides an approach to more effectively leveraging the continuous improvements to SAT solvers that have been advancing the field of automated software verification over the last decade.

Encouraged by this achievement, Chapter 5 turned to the question on how partial-order encodings interact with the underlying decision procedures in SMT solvers. For this, we studied a simple, yet relevant and challenging, concurrency problem. This problem instance turned out to be important from a proof-theoretical point of view due to its exponential-sized Fixed-Alphabet DPLL($\mathcal{T}$) proof complexity (Figure 5.3). Crucially, our experimental and theoretical results revealed the importance of the selection and value constraints ($\mathcal{T}_S + \mathcal{T}_V$) in understanding this exponential proof complexity. This marked a paradigm shift in the study of partial-order encodings because previously the focus in the literature had been primarily on the partial-order constraints $\mathcal{T}_C$ rather than $\mathcal{T}_S + \mathcal{T}_V$. To start address this newly discovered issue, we experimented with $\mathcal{T}_V$-lemmas in the form of unbounded intervals. In the restricted case of linear arithmetic theory, we showed that these interval theory lemmas can reduce a huge part of the search space, providing a concrete direction for future research. Contrary to our experiments with the supremum variables in the quadratic-size partial-order encoding, our experiments with interval theory lemmas show that adding variables to an encoding can be advantageous in certain cases. Importantly, the results of our work firmly established the concurrency challenge problem, and strict generalizations thereof (see next section), as the basis for experimentally evaluating future developments in this research area.

The problems uncovered by Chapter 5 made it plausible to ask what happens if we step outside the confines of a SAT/SMT solver. More accurately, the research question we had in mind is whether or how the insights of SAT/SMT solvers can be transfered to other domains. After all, NP-complete problems are provably polynomial-time reducible to each other. In particular, we sought a form of non-chronological backtracking for the NP-complete linearizability checking problem. By manually analyzing conflicts encountered by Lowe's linearizability checker [Low15], we identified the

concept of P-compositionality as a way of avoiding certain conflicts altogether. Our investigation resulted in the first experimental evaluation of a partitioning scheme for checking the linearizability of associative concurrent data types, such as lock/wait-free sets and maps. We showed that our technique improves the state-of-the-art linearizability checker in the WGL-family by one order of magnitude in time and/or space. One aspect that makes the linearizability checkers in this family particularly attractive from a verification perspective is the fact that they deal with the specification of the concurrent data type as executable code rather than a logical formula. We believe that there is merit to run-time verification algorithms that can capture different instances of the same problem by calling a user-supplied program because this appeals to practitioners who may not be familiar with more traditional formal verification techniques. Of course, in the case of our linearizability checker, this came at the cost of limited automated reasoning power about non-deterministic data inputs.

In summary, this thesis systematically studied a range of automated techniques for finding concurrency-related bugs using partial-orders. Among these, we showed how to gain new insights into the state-of-the-art partial-order encoding by separating it into three separate theory-specific parts, namely $\mathcal{T}_C$, $\mathcal{T}_S$ and $\mathcal{T}_V$. This separation led to a new partial-order encoding that can drastically reduce the run-time of the analysis prior to calling the SAT/SMT solver, and new theoretical and experimental results on how partial-order encodings interact with backtracking algorithms during SAT/SMT solving. This finding stimulated a novel partitioning scheme that was shown to significantly improve the state-of-the-art linearizability checker.

7.1 Future work

There are many avenues for further work in both theoretical and practical nature.

Quiescent consistency checker Quiescent consistency, which is a weaker notion of linearizability, also satisfies the locality principle [HS12, p. 51]. It would therefore be interesting to derive from quiescent consistency a new conflict and backtracking avoidance scheme that is more general than P-compositionality.

Dynamic partial-order encoding techniques The OpenCores Ethernet MAC had to be manually extracted from QEMU to omit QOM-specific features because they are infeasible to symbolically analyze. To avoid this manual effort, the next step is to combine the symbolic analysis with run-time information that can be obtained

from the concrete execution of the program. In the symbolic execution literature, this has shown to be a promising approach for sequential programs, e.g. [GKS05, CKC11].

Read-from summaries Our theoretical and experimental results suggest that a more incremental approach to partial-order encodings may be needed to overcome the fixed alphabet problem in DPLL($\mathcal{T}$). This may involve inferring constraints on demand as the SMT solver explores different possibilities for an interpretation of the 'read-from' function. We are investigating this in the form of 'read-from' summaries that are expressed with respect to clock constraints and lazily introduced variables that constrain the values in memory.

Decidability of DPLL($\mathcal{T}$) alphabet extension for partial-order encodings It is unclear under what condition an DPLL($\mathcal{T}$) alphabet can be automatically extended such that a polynomial-size DPLL($\mathcal{T}$) proof of unsatisfiability can be found for partial-order encodings and how this problem may be related to backdoor variables in propositional logic [GKSS08].

Abstract domains in DPLL($\mathcal{T}$) Another related research direction is to further explore the capabilities of recent extensions of DPLL($\mathcal{T}$) that are not restricted to a fixed alphabet. As shown by our experiments with ACDCL and MCSAT in Section 5.7, it is not clear what is exactly needed to provide the necessary expressiveness for learning facts about value constraints. With our simplified version of the concurrency challenge problem, we showed that the problem is not specific to partial-order encodings but it is likely that inferred value constraints need to be expressed with respect to clock constraints. To see this, consider a strict generalization of our concurrency challenge problem in which each thread $\mathtt{T}_k$, for $1 \leq k \leq N$, increments the value at memory location x by k rather than always one. Given such a concurrent system, the challenge is to automatically prove $[x] \leq \frac{N(N+1)}{2}$ for a fixed integer $N > 1$. Note that this simple change seems to break the symmetry that is present in the concurrency challenge problem from Figure 5.1.

Partial-order encodings for abstract data types To date, partial-order encodings are predominately used for modelling shared memory reads and writes. It would be interesting to model higher-level operations as well. This could include the operations on concurrent data structures such as maps and sets. If the implementation of these data structures can be assumed to be correct, this would likely help scale

the analysis to larger programs because the low-level reads and writes required to implement the data structure can be omitted from the analysis.

Supremum constraints Recall that at the core of the quadratic-size partial-order encoding is the idea of finding the supremum of certain sets. In our SAT-based implementation, we symbolically encode this using fixed-size bit-vector inequality constraints (see Appendix E). It could be that the solving of these constraints would work better with the backtracking search in SAT solvers if we take into consideration the effects of unit propagation. In particular, it would be interesting to consider a supremum encoding which introduces extra 'carry' bits so that the SAT solver can earlier detect conflicts.

Loops All current partial-order encodings are only applicable for symbolic techniques that are bounded up to a certain depth. It is still an open problem how to handle loops other than by unwinding them a finite number of times. This problem may also provide potential applications of our SMT-based formalization of the partial-string refinement problem in Chapter 3 since it makes the first steps towards automatically reasoning about two partial-order abstractions of a concurrent system, which may be helpful in inferring invariants of value constraints with respect to partial-order constraints.

Appendix A

Example of memory reordering

To illustrate the effect of weak memory reorderings, consider the small concurrent system in Figure A.1. It consists of two threads, $\mathtt{T}_1$ and $\mathtt{T}_2$, that access two distinct shared memory locations x and y. The memory at location x is denoted by $[x]$. Assume both memory locations are initialized to zero, i.e. initially $[x] = [y] = 0$. The threads perform the following store and load instructions: $\mathtt{T}_1$ writes '1' into memory location x and then reads the value at memory location y into register v_1, whereas $\mathtt{T}_2$ writes '1' into memory location y and then reads the value at memory location x into register v_2. Note that registers are always local to a thread and cannot be shared. If we enumerate all interleavings of $\mathtt{T}_1 \parallel \mathtt{T}_2$, we can see that after both threads have terminated the two registers cannot both contain the value zero; equivalently, $v_1 = v_2 = 0$ is impossible under interleaving semantics. By contrast, weak memory semantics, such as TSO, can permit $v_1 = v_2 = 0$ as final register values. Informally, this computation can occur when the store and load instruction in each thread are reordered or the stores to memory are buffered. Note that these kind of phenomena are not unique to weak memory architectures but can also be observed in certain weakly consistent distributed systems.

Thread $\mathtt{T}_1$		Thread $\mathtt{T}_2$
$[x] := 1$	$\parallel$	$[y] := 1$
$\mathbf{local}\, v_1 := [y]$	$\parallel$	$\mathbf{local}\, v_2 := [x]$

Figure A.1: A program implementing the classical idiom of Dekker's algorithm.

Appendix B

Mathematical background

This appendix recalls elementary concepts from lattice and order theory (e.g. [DP02]) that we use for Chapter 3 and 4.

Denote the set of **natural numbers** by $\mathbb{N} = \{1, 2, \ldots\}$. The **powerset** of a set P is the set of all subsets of P, denoted by $\mathcal{P}(P)$. The **empty set** is denoted by $\emptyset$. We write "$\triangleq$" for definitional equality. The **Cartesian product** $X \triangleq X_1 \times \ldots \times X_n$ of sets $X_1, \ldots, X_n$ is defined to be the set of all ordered n-tuples $\langle x_1, \ldots, x_n \rangle$ with $x_1 \in X_1, \ldots, x_n \in X_n$. Two elements $\langle x_1, \ldots, x_n \rangle$ and $\langle y_1, \ldots, y_n \rangle$ of X are defined to be **point-wise equal** whenever the coordinates x_i and y_i are equal for each $1 \leq i \leq n$.

Let P be a set. A **binary relation on** P is a subset of $P \times P$. A **preorder** is a binary relation $\preceq$ on P that is **reflexive** ($\forall x \in P \colon x \preceq x$) and **transitive** ($\forall x, y, z \in P \colon (x \preceq y \wedge y \preceq z) \Rightarrow (x \preceq z)$). We write $x \npreceq y$ when $x \preceq y$ is false. For all $x, y \in P$, $x \prec y$ is called **strict** and is equivalent to $x \preceq y$ and $y \npreceq x$. A **partial order** is a preorder $\leq$ that is **antisymmetric** ($\forall x, y \in P \colon (x \leq y \wedge y \leq x) \Rightarrow (x = y)$). By reflexivity, partial orders satisfy the converse of the antisymmetry law, i.e. $\forall x, y \in P \colon (x \leq y \wedge y \leq x) \Leftrightarrow (x = y)$.

A **partially ordered set**, denoted by $\langle P, \leq \rangle$, consists of a set P and a partial order $\leq$. Every logical statement about a partial order $\langle P, \leq \rangle$ has a **dual** that is obtained by using $\geq$ instead of $\leq$. A **minimal element** $x \in P$ satisfies $\forall y \in P \colon y \leq x \Rightarrow x = y$. Dually, a maximal element $x \in P$ satisfies $\forall y \in P \colon x \leq y \Rightarrow x = y$. For all $Q \subseteq P$, $\uparrow_{\leq} Q \triangleq \{y \in P \mid \exists x \in Q \colon x \leq y\}$ is the **upward-closed set** of Q in $\langle P, \leq \rangle$. As expected, the dual is called the **downward-closed set** of Q, denoted by $\downarrow_{\leq} Q$. Usually, we write $\uparrow Q$ instead of $\uparrow_{\leq} Q$ when the ordering is clear. Abbreviate $\uparrow x \triangleq \uparrow \{x\}$. For all $H \subseteq P$, $x \in P$ is an **upper bound** of H if $y \leq x$ for all $y \in H$; x is called the **least upper bound** (or **supremum**) of H, denoted by $\bigvee H$, if x is an upper bound of H and if, for every upper bound y of H, $x \leq y$. By antisymmetry,

$\bigvee H$ is unique, if it exists. The **lower bound** and **greatest lower bound**, written as $\bigwedge H$ where $H \subseteq P$, are defined dually. The (unique) **least element** in P, if it exists, is $\bot \triangleq \bigvee \emptyset$ whose dual, if it exists, is $\top \triangleq \bigwedge \emptyset$. A **lattice** is a partial order $\langle L, \leq \rangle$ where every two elements have a (necessarily unique) least upper bound and greatest lower bound: for all $x, y \in L$, these are denoted by $x \vee y$ and $x \wedge y$, respectively. A **complete lattice** $\langle L, \leq, \wedge, \vee, \bot, \top \rangle$ is a lattice where $\bigvee S$ and $\bigwedge S$ exists for every $S \subseteq L$.

Let $\langle P, \leq \rangle$ and $\langle Q, \sqsubseteq \rangle$ be partial orders. A function $f \colon \langle P, \leq \rangle \to \langle Q, \sqsubseteq \rangle$ is called **monotonic** exactly if $\forall x, y \in P \colon x \leq y \Rightarrow f(x) \sqsubseteq f(y)$. Given three sets P, Q and R, the **composition** of two functions $f \colon P \to Q$ and $g \colon Q \to R$, denoted by $g \circ f$, is a function from P to R such that $g \circ f(x) \triangleq g(f(x))$ for all $x \in P$.

Appendix C

SMT Kit: A C++11 library for many-sorted first-order logic

C.1 Introduction

In this appendix, we explain the technical considerations that have gone into the design of a new library for quantifier-free, many-sorted first-order logic. This library helped us generate symbolic expressions as part of our experimental research into exponential-sized DPLL($\mathcal{T}$) proofs (Chapter 5). Since many SMT solvers provide a programmatic C and/or C++ interface, our library is written in a recently standard version of of the C++ programming language, namely C++11 [ISO12].

The standard way of interfacing with SMT solvers is through SMT-LIB [BFT15], a standard that is carefully defined in terms of *many-sorted first-order logic*. This means that each SMT-LIB expression is associated with a so-called sort. In many ways, a *sort* is similar to a type. For example, a sort could be a fixed-size bit vector of length eight, i.e. a byte. This is illustrated by the quantifier-free formula in Figure C.1 where x and y denote arbitrary 8-bit unsigned integers and `0xF7u` is an 8-bit unsigned hexadecimal literal.

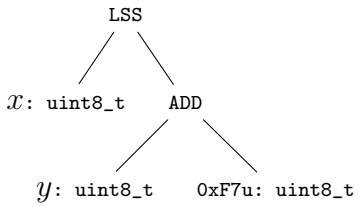


Figure C.1: Symbolic expression of "x `<` `(`y `+ 0xF7u)`" including sort information.

One feature that makes SMT-LIB so interesting is that sorts can be mixed in non-trivial ways. For example, we have already seen through the experiments in **??** that the sorts of clock and value constraints can be mixed. This is motivation to give sorts a prominent role when designing a library for many-sorted logics. In particular, our goal is to treat information about sorts statically, whenever possible, so that we can leverage the compiler's built-in type checker. To achieve this, we turn to template metaprogramming [Vel96] because it allows us to write programs that run at compile-time. This opens up a new perspective on how to implement the following library features:

1. Abstract data types for a quantifier-free, many-sorted first-order logic expressions with support for CVC4, MathSAT5 and Z3 (Section C.2);

2. Type-dependent maximal expression sharing that can avoid possibly error-prone and expensive dynamic casts (Section C.3);

3. A code generator that can simplify symbolic expression according to the mathematical properties of an operator and its operands (Section C.4);

We emphasize that our focus is not so much on the features themselves but how we can leverage the compiler for more convenient type inference and type checking purposes.

C.2 Statically-typed symbolic expressions

Recall that we want to implement symbolic expressions such that we can infer their type using standard C++11 language features. However, even without type inference, there is no standard implementation of symbolic expressions, and two conceivable designs may be best illustrated by existing tools such as CBMC [KT14] and KLEE [CDE08]:

1. CBMC uses one designated data structure, called `irept`, where each node in the expression tree stores its children in a dynamically allocated array (`std::vector`) or symbol/hash table;

2. KLEE, on the other hand, is inspired by the expression tree design in LLVM which has many specialized in-memory data structures for each kind of expression node including constants, compare instructions etc.

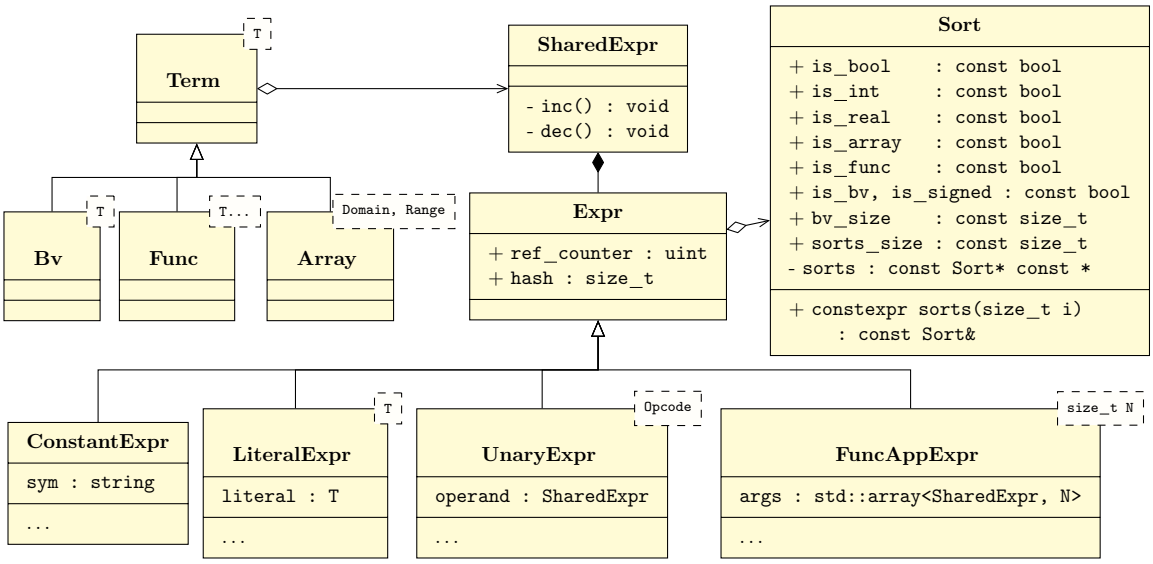


Figure C.2: High-level UML class diagram of C++11 library for many-sorted first-order logic (not all classes are shown).

Of course, what these or similar designs have in common is that structurally equal expressions can be *shared*, a feature that our library supports by the standard means of pointers and reference counters (see also `SharedExpr` smart pointer and `Expr` class in Figure C.2).

What sets our design of symbolic expressions apart, however, is that it is tailored towards many-sorted first-order logics. In particular, we target the quantifier-free first-order logics defined in SMT-LIB [BFT15], including sorts such as fixed-size bit vectors, mathematical integers and arrays.

Our library retains information about sorts through a template class `Term<T>` where objects of type `Term<T>` represent shared sorted symbolic expressions whose sort is `T`. However, code that directly builds expressions from `Term<T>` would quickly become unwieldy. We therefore rely on the curiously recurring template pattern (CRTP) [Cop96] to have a subclass of `Term<T>` for each sort. For example, the shared expressions in Figure C.1 are of type `Bv<uint8_t>` because they are bit-vectors of length eight. The relative position of `Bv<uint8_t>` in the class hierarchy is illustrated by Figure C.2. In particular, the fact that we rely on CRTP means that `Bv<uint8_t>` derives from `Term<Bv<uint8_t>>`. In other words, `Bv<uint8_t>` is a subclass of a base class, `Term<T>`, whose template parameter is instantiated to the derived class itself. This may seem a bit circular but is in fact perfectly valid C++98 code. Analogously, we declare `Bool`, `Int`, `Real`, `Func<T...>` and `Array <Domain, Range>` to correspond to Booleans, mathematical integers, reals, uninterpreted functions and McCarthy arrays, respectively.

Before the advent of C++11, however, this basic design would have been too restricted to make a practical difference in the design of a many-sorted first-order logic library. What changes this with C++11 are at least five new language features:

1. *Compile-time constant expressions* can be built via the `constexpr` keyword;
2. *Variadic template parameters* `T...` can represent an arbitrary number of type as well as certain non-type parameters in a type-safe manner;
3. The `decltype` keyword and *type traits* give the means to query the type of an expression, or its characteristics, at compile-time;
4. *Perfect forwarding* enables a function f to call another function g with f's arguments such that their value categories are preserved;
5. *Trailing-return-types*, denoted by `->` after a function signature, allow the return type in function declarations to be inferred.

These new C++ language features — or many of the new and existing standard C++11 libraries that built on them — allow programs to be much more tightly integrated with the C++ compiler. SMT Kit applies these C++11 features as follows:

1. We define `Sort` to be an immutable class (see Figure C.2) that is instantiated at compile-time with (possibly recursive) sort information about SMT-style expressions;
2. For sorts such as `Func<T...>` and `Array<Domain, Range>`, we traverse the list of (variadic) template arguments with a mutual recursive template metaprogram;
3. We overload operators together with perfect forwarding and trailing-return-types to efficiently construct symbolic expressions at run-time.

Throughout the rest of this section, we discuss the relevant implementation details and highlight the significance of template metaprogramming as a way to make a quantifier-free fragment of many-sorted logics transparently usable through a (mostly header-only) C++11 library.

To motivate the technical discussion we are about to embark, recall that our C++11 library features (among several others) fixed-size bit vectors and uninterpreted function sorts of arbitrary arity (subject to finite compiler resources). For example, `Decl<Func<Bv<uint16_t>, Bv<uint8_t>>> f` says that `f` is an uninterpreted unary function that maps a 16-bit word to a single byte, cf. `declare-fun` in [BFT15, §4.2.3]. In

fact, since we are overloading operators, an end-user of our library with a C++11-compliant compiler will be able to write equalities such as `f(x) == y` where `y` has type `Bv<uint8_t>` — as in fact enforced by the type checker. But how are sorts of these kinds of expressions inferred and converted to `Sort` objects?

To answer this, we look at a template metaprogram called `SortSwitch` that takes as input a type `T` and returns as output a statically allocated `Sort` object for `T`. If we had only a single sort such as bit vectors, the following code would suffice:

```
template<class T>
struct BvSort
{
  static constexpr Sort s_sort = Sort(/* type traits of T */);
};

/* Fixed-size bit vector */
template<class T>
struct SortSwitch
{
  typedef BvSort<T> Type;
};
```

This template declaration will be used to write a template metaprogam, `SortSwitch`, that can encode a switch statement that runs at compile-time. For example, the template instantiation `SortSwitch<uint8_t>::Type::s_sort` calls the template metaprogam with `uint8_t` as an argument and the return value is a statically allocated `Sort` object that represents a fixed-size 8-bit vector sort. The instantiation of `Sort` is accomplished via standard C++11 type traits such as `std::is_integral<T>` and `std::is_signed<T>` whose values determine the initialization of the `Sort` member fields (e.g. `Sort::is_bv`) shown in Figure C.2. To populate the other member fields, particularly `sorts`, we resort to pattern matching on the template parameter `SortSwitch<T>`. Of course, in functional programming, pattern matching is a standard way to define inductive types of which, in fact, `Sort` is as an example.

In C++, a form of pattern matching on types is possible through so-called *template specializations* as illustrated next:

```
/* Uninterpreted function */
template<class... T>
struct SortSwitch<Func<T...>>
{
  typedef Math<Func<T...>> Type;
};
```

This template specialization means that `SortSwitch<U>` instantiates `Math<Func<T...>>` whenever `U` is `Func<T...>`. An example of such an instantiation is `SortSwitch<F>::Type` where `F` might represent an uninterpreted function `Func<Bv<uint16_t>, Bv<uint8_t>>`. It follows the

code for `Math<Func<T...>>`, which itself is a template specialization:

```
template<class T, class... U>
struct Math<Func<T, U...>>
{
  static constexpr size_t N = sizeof...(U) + 1;
  static constexpr Sort s_sort = Sort(/* type traits ... */
    FuncSort<SortArray<N, &SortSwitch<T>::Type::s_sort>,
      U...>::result());
};
```

It helps to unravel each line of code. The first line says that `T` is a type and `U...` denotes a variable-length list of types. The intuition here is that `T` is the sort of the first argument of an uninterpreted function whereas `U...` are the sorts of the remaining function arguments including the sort of its codomain. Next `N` should be interpreted as the function arity, e.g. a binary function has `N` equals two. The last two lines show that `Math<Func<T, U...>` is a mutual recursive template metaprogram because it calls `SortSwitch<T>` to instantiate a `Sort` object. At last we define `SortArray` and `FuncSort` each in turn:

```
/* Aggregate a compile-time constant expression
 * that is an N-sized array of Sort pointers */
template<size_t N, const Sort* const... sorts>
struct SortArray {};
```

This is the first template metaprogram that we have encountered so far whose template parameters are not types but an integral constant expression, `N`, that corresponds to the length of the variadic list of pointers, `sorts...`. The goal of `FuncSort` is to recursively construct this list at compile-time. Here is the base case of this construction:

```
template<class... T> struct FuncSort {};

/* Base case for uninterpreted function sort */
template<size_t N, const Sort* const... sorts>
struct FuncSort<SortArray<N, sorts...>>
{
  static constexpr const Sort* const s_sorts[N] = {sorts...};

  static constexpr const Sort* const (&result())[N]
  {
    return s_sorts;
  }
};
```

Note that the template specialization of `FuncSort<T...>` has the same argument list as `SortArray`. In fact, `sorts...` and its length `N` are merely extracted from `SortArray`, as done by the statically evaluated member function `FuncSort<T...>::result`. In other words, `SortArray` serves as an auxiliary container data structure that aggregates an array of `Sort` pointers that can be extracted by calling the static member function `FuncSort<T...>::result`. To

see how this works recursively, we give the last template specialization of `FuncSort`:

```
template<class T, class... U, size_t N, const Sort* const... sorts>
struct FuncSort<SortArray<N, sorts...>, T, U...>
{
  /* prepend sort for T, recurse on U... */
  typedef FuncSort<SortArray<N, sorts...,
    &SortSwitch<T>::Type::s_sort>, U...> Build;

  static constexpr const Sort* const (&result())[N]
  {
    return Build::result();
  }
};
```

We can see now that `FuncSort` is a template metaprogram with possibly two recursive calls: (1) `FuncSort` calls itself but (2) mutual recursion may also occur through the template instantiation `SortSwitch<T>::Type::s_sort`. However, in the case of SMT-style expressions, the second recursion can be ruled out because there are no higher-order functions in many-sorted first-order logic. Our implementation though relies on calling conventions and the various run-time checks in the SMT solvers rather than explicitly enforcing this restriction at compile-time. By design, the underlying template metaprograms are largely transparent in the library and we may simply call `internal::sort<T>()` to statically compute the `Sort` object of a sort `T`. From now on, therefore, we simply assume that we have been given the right `Sort` object.

The last main question to answer then is how to use expressions of a given sort. Fortunately, this is comparatively easy by overloading operators. We remark though that this is only straightforward when class constructors are designed such that operator overloading ambiguities are avoided. This extra complexity arises as we support both strongly-typed expressions such as `Bv<uint8_t>` and a type-unsafe version thereof through `SharedExpr`, see also Figure C.2.

To illustrate operator overloading for uninterpreted function applications, such as `f(x)` where `f` is an uninterpreted function, consider the following template member function declaration in a template class that represents uninterpreted functions (we have omitted the full class definition for brevity):

```
template<class... Args>
typename Func<T...>::Range operator()(Args&&... args) const;
```

Note that the overloaded function call operator involves the template metaprogram `Func<T...>::Range` that is defined to return the last type in the sequence `T...`, i.e. the codomain of the uninterpreted function. The domain, in turn, is modelled by `Args`, and these arguments are statically type-checked and converted to a fixed-size array by another template metaprogram that uses a recursive technique similar to the one

Arithmetic	Relational	Logical	Bitwise	Other
`a + b`	`a == b`	`!a`	`~a`	`a[b]`
`a - b`	`a != b`	`a && b`	`a & b`	`*a`
`+a`	`a > b`	`a || b`	`a | b`	`&a`
`-a`	`a < b`		`a ^ b`	`a->b`
`a * b`	`a >= b`		`a << b`	`a->*b`
`a / b`	`a <= b`		`a >> b`	`f(a, b)`
`a % b`				`a, b`
`++a`				`(T) a`
`a++`				`new T`
`--a`				`new T[n]`
`a--`				`delete a`
				`delete[] a`

Table C.1: C++ operators overloading.

found in `std::make_index_sequence`, a recent C++14 addition to the `<utility>` header [ISO14]. Given an array of say `N` uninterpreted function arguments, the overloaded call operator for uninterpreted function applications instantiates then at run-time the corresponding `FuncAppExpr<N>` expression depicted in Figure C.2. This design is plausible because there is no function currying in SMT-LIB. For compatibility reasons, we also support `apply` as an external template function with the same effect as the overloaded call operator.

C.3 Type-dependent hash consing

Our repertoire so far includes type-safe (or optionally not) data types of symbolic expressions. And where there are expressions, there is hash consing [Ers58, Got74] — a standard dynamic programming technique that relies on hash tables to achieve maximal sharing of expressions that are structurally equal, e.g. [CDE08, BH08, BGM13, KT14]. If done right, hash consing can decrease memory footprints but more importantly it guarantees constant-time structural equality checks. The fact that our library adopts template metaprogramming changes how we choose to implement hash consing. In particular, it is natural to consider hashing expressions according to their static type. The benefits of such a design are threefold:

1. Hash functions can make stronger assumptions about the kind of objects in the hash table;

2. Member functions on expressions in the hash table may be called without any dynamic casts that could contribute to run-time overhead or failures;

3. Hash consing can be statically yet selectively turned on and off for each type of expression.

The bottom part of Figure C.2 sketches the class hierarchy of expressions whose objects we want to hash. We emphasize that an *expression* is an object of a subclass `Expr`, whereas a *shared expression* is an object of type `SharedExpr`. The difference is that the former is a vertex in a directed-acyclic graph whereas the latter is a reference-counted pointer to such a vertex.

With this distinction in mind, we declare `make_shared_expr` to be a template function with external linkage. Hence C++ guarantees that static local variables are unique for each template function instantiation. Therefore `make_shared_expr` may store a pointer to a newly allocated expression in a static local hash table prior to returning this pointer back to the caller.

However, this basic design would not yet account for the case when a reference counter of an `Expr` decreases to zero. To remedy this, we recall the concept of a *weak hash table*: a hash table that ensures that its keys are deleted as soon as they are not referenced anymore from outside the hash table. Unlike approaches such as in [FC06], the challenge here is that C++ provides no automatic garbage collection. This suggests *weak pointers* as a way to satisfy the basic requirement that the memory of unused symbolic expressions is freed. However, a more sophisticated design is needed to also guarantee that an associated key in the hash table is deleted. For this, it is still possible to reuse existing container data structures in the C++ standard library but we have to be aware of the following design restrictions and/or trade-offs:

- `std::set` is an ordered set that is usually implemented as a red-black tree where iterators are stable under all essential operations (search, removal, and insertion) with the drawback that they have logarithmic time complexity;

- By contrast, operations on `std::unordered_set` have (amortized) constant-time complexity but iterators are unstable and their pointed to element is only accessible through a constant reference to prevent data changes that could otherwise invalidate the hash key;

- Thus, the element type in `std::unordered_set` would have to be a shared expression, but as soon as the destructor of the pointed to (possibly mutable) expression is called, it could not remove itself from the hash table because the C++11 container interface would force it to call the hash function of an object undergoing destruction, which would lead to undefined behaviour;

- It is therefore tempting to map hash values of expressions to shared expressions; in general, however, neither `std::map` nor `std::unordered_map` are suitable due to the possibility of hash collisions, i.e. equal keys.

Therefore, if we want to achieve automatic deletion of key-value pairs with (amortized) constant-time complexity, we have to resort to `std::unordered_multimap`, or another implementation to the same effect. In this setting, the key is the hash value of an expression E and the value is a pointer to E. The fact that the values in the hash table are shared expressions instead of expression objects goes back to the need to erase the key-value pair associated with an expression undergoing deletion. Alternatively, we could exploit the fact that iterators in `std::multimap` are stable; but the downside is that there is no natural ordering between expressions and we could only achieve logarithmic time complexity.

We therefore opt for `std::unordered_multimap`; the relevant code though can be optionally disabled via a preprocessor flag. If this flag is enabled, hash consing automatically occurs every time the template function `make_shared_expr` is called. Since `make_shared_expr` is a template function that takes a variable-length list of arguments of which an expression is comprised, we can even check whether there already exists a structurally equal expression without having to instantiate a temporary expression. Moreover, `make_shared_expr` is designed with exception-safety in mind to avoid memory leaks etc.

In our library, `Expr` is the base class for every expression. As usual, objects of type `Expr` are reference counted. This is done in `SharedExpr`. Once the reference counter in an `Expr` object becomes zero, its destructor is called. Importantly since the destructor of every `Expr` object is virtual, the destructor of any of its subclasses is called first. This is where the hash table for hash consing comes in: when the destructor of an `Expr` subclass is called (e.g. `~LiteralExpr`), it should delete itself from the hash table. This means that our hash tables only store expressions whose reference counter is positive. An implementation that guarantees this invariant, however, is more intricate and deserves a closer look.

C.4 Symbolic expression simplifications

In practice, it is often convenient to simplify symbolic expression before they are handed to the SMT solver. For example, an arithmetic expression such as $x + 2 + 3$ should be simplified to $x + 5$ whenever possible. Here is where C++11 template metaprogramming can make another difference: we can statically generate code that builds and simplifies symbolic expressions at run-time according to the mathematical

properties of the operator and its operands. Through this mechanism, it is possible compute a simpler symbolic expression once at compile-time and re-use it at run-time, or selectively apply symbolic expression simplifications that are deemed important for certain operators but not for others.

As a first step, we let `Opcode` be an enumeration type that enumerates all operators that can be overloaded. For the most part, all of the operators given in Table C.1 can be overloaded, as applicable. This includes unary and binary arithmetic operators (e.g. `!`), binary relations (e.g. `!=`) and bitwise operators (e.g. `<<`). For example, `LNOT` has type `Opcode`, and as the name suggests, `LNOT` represents logical negation. C++ overloading is also supported for (compound) assignment statements, e.g. `a = b` and `a += b`. To give a first flavour of our simplification procedure, we declare the following template class:

```
template<Opcode opcode> class Eval;
```

The following extract of the template class specialization shows how one could generate code for the unary negation operator:

```
template<>
struct Eval<LNOT>
{
  template<class U>
  static inline auto eval(U&& u) -> decltype(!std::forward<U>(u))
  {
    return !std::forward<U>(u);
  }
};
```

This template metaprogram might look unfamiliar because it uses C++11 automatic type inference features, universal references [Mey12] and perfect forwarding.[1] Informally, these features instruct the compiler to generate optimal code in the sense that memory copy operations are avoided whenever possible. For our high-level discussion here, it is sufficient to appreciate that calls such as `Eval<opcode>::eval(x)`, where `opcode` has type `Opcode`, statically generates the code that calls the right unary operator with `x` as argument. Similarly, we can write `Eval<opcode>::eval(x, y)` for a binary operator. In simple cases, a macro could achieve the same thing except that it would be expanded by the preprocessor without any type checking. This difference is important for more complicated transformations. In particular, template metaprogams shine when we want to generate code that simplifies symbolic expressions according to operator-specific rewrite rules, as shown next.

Before the advent of C++11, it would have been difficult to generate code for

[1] Since C++14 [ISO14], `std::forward` has also `constexpr` support.

expression simplifications in a portable way. For example, $x+2+3 = x+5$ for unsigned integers even when there is overflow, whereas this equality does not generally hold when it involves signed integers. More generally, the goal is to identify mathematical properties of operators and their operands at compile-time. This problem could be framed as identifying algebraic properties of operators. For example, addition on unsigned integral types with 0 as identity element forms an algebraic structure known as a commutative monoid. The following template metaprograms illustrates how this algebraic structure can be statically identified by reusing the compiler's type checker:

```
template<Opcode opcode, class T>
struct IsCommutativeMonoid :
  std::integral_constant<bool,
    std::is_integral<T>::value and
    std::is_unsigned<T>::value and
    (ADD == opcode /* or ... */)> {};
```

For example, `IsCommutativeMonoid<ADD, float>` is the template instantiation of a class that inherits from the class `std::false_type` because addition on floating point numbers is generally not associative.[2] By contrast, `IsCommutativeMonoid<ADD, unsigned int>` inherits from `std::true_type`. This can be used to statically enable or disable expression simplifications depending on the mathematical properties of the overloaded operator and its operands.

To see how this works, consider two non-template classes `Lazy` and `Eager`. Both classes are defined to have a static template function called `simplify`. The intention is that `Lazy::simplify` rewrites symbolic expressions by propagating one or both of its arguments if there are literals, whereas `Eager::simplify` does not. We can then conditionally invoke either `Lazy::simplify` or `Eager::simplify` as follows:

```
return std::conditional<
/* if */ IsCommutativeMonoid<opcode, T>::value,
/* then */ Lazy,
/* else */ Eager>::type::template simplify<opcode>(lhs, rhs);
```

where `opcode` has type `Opcode`, and `T` is the type of two operands denoted by `lhs` and `rhs`. Since this conditional will be evaluated at compile-time, it can be seen as a type-specific way of generating code for a symbolic interpreter that statically takes into account the algebraic properties of the operator. Similar techniques can be used to generate code of a symbolic interpreter that simplifies at run-time Boolean expressions such as **false** && x and **true** || x where `x` is a Boolean constant and **false** and **true** act as annihilators in logical conjunctions and disjunctions, respectively.

[2] In general, floating point arithmetic does not even have a unique identity element unless the rounding mode is fixed throughout the lifetime of the program.

C.5 Concluding remarks

We have explained how to generate symbolic expressions by leveraging (mutual) recursive template metaprograms, operator overloading and recently standardized C++11 language features such as perfect forwarding, trailing-return-types, and type traits.

C.6 Bibliographic notes

Our work is closely related to PySMT [GM15]. Since this library is written in a type-unsafe language, however, it cannot offer any type guarantees. Yet another library is SBV written in Haskel.[3] In contrast to both PySMT and SBV, our library can be more directly integrated with the C and C++ interface of mainstream SMT solvers.

More broadly, the work discussed in this appendix is also related to tools such as CBMC [KT14] and KLEE [CDE08] as already discussed in C.2. These tools belong to the family of algorithms that construct symbolic expressions to compute the strongest postcondition of a program under scrutiny. This is commonly called symbolic execution. Despite its simplicity—or perhaps because of it—symbolic execution is an effective and surprisingly versatile program analysis technique, e.g. [GKS05, CDE08, Ana14]. Our library design could be seen as the basic ingredient for implementing symbolic execution of programs that only invoke operators that can be overloaded. Such an approach could be seen complementary to tools that symbolically execute programs by instrumenting source code, e.g. [KPV03, VPK04, BS08]. One difference is that tools such as JPF-SE [KPV03] and later extensions [VPK04] have to take into account Java's type erasure, whereas C++ template metaprogramming is Turing-complete. CREST [BS08] is another interesting tool that performs a source-to-source transformation on the program under scrutiny to emulate a stack machine.

[3] http://leventerkok.github.io/sbv/

Appendix D

Concurrent Kleene algebra of partial strings

This appendix fills in details for Section 3.3 where we briefly summarized known algebraic properties of certain downward-closed sets of partial strings.

Recall that we specifically designated the following two sets of finite partial strings: $0 \triangleq \emptyset$ and $1 \triangleq \{\bot\}$ where $\bot$ is the empty partial string. It is easy to see that both 0 and 1 are closed under the downward-closure with respect to $\sqsubseteq$, and therefore they are programs (Definition 3.3.1).

Proposition D.0.1. $\downarrow_{\sqsubseteq} 0 = 0$ *and* $\downarrow_{\sqsubseteq} 1 = 1$ *in* $\mathbb{P}$. *So* 0 *and* 1 *are programs.*

Proof. Clearly $\downarrow_{\sqsubseteq} 0 = \downarrow_{\sqsubseteq} \emptyset = \emptyset = 0$. By Definition 3.3.1, $1 = \{y \in \mathsf{P}_f \mid y \sqsubseteq \bot\}$. Let $y \in \mathsf{P}_f$ such that $y \sqsubseteq \bot$. By Definition 3.2.1 of the empty partial string and Definition 3.2.6, there exists a bijective monotonic morphism $f \colon \bot \to y$. Since $\bot$ is unique, $y = \bot$. We conclude that $\downarrow_{\sqsubseteq} 1 = 1$. □

Proposition D.0.2 (Annihilator)**.** *For every program* $\mathcal{X} \in \mathbb{P}$, $\mathcal{X} \bowtie 0 = 0 \bowtie \mathcal{X} = 0$.

Proof. Immediate from $0 \triangleq \emptyset$ and Definition 3.3.5. □

Equivalently, by our "bow tie" convention, we get the following two statements: $\mathcal{X} \parallel 0 = 0 \parallel \mathcal{X} = 0$ and $\mathcal{X}; 0 = 0; \mathcal{X} = 0$ for every program $\mathcal{X}$.

Proposition D.0.3 (Identity)**.** *For every program* $\mathcal{X} \in \mathbb{P}$, $\mathcal{X} \bowtie 1 = 1 \bowtie \mathcal{X} = \mathcal{X}$.

Proof. Immediate from Proposition 3.2.11. □

Proposition D.0.4 (Distributivity)**.** *For every program $\mathcal{X}$ and family of programs $\{\mathcal{Y}_n \mid n \in \mathbb{N}\}$, the following equalities holds:*

$$\mathcal{X} \bowtie \left(\bigcup_{n\geq 0} \mathcal{Y}_n\right) = \bigcup_{n\geq 0} (\mathcal{X} \bowtie \mathcal{Y}_n)$$
$$\left(\bigcup_{n\geq 0} \mathcal{Y}_n\right) \bowtie \mathcal{X} = \bigcup_{n\geq 0} (\mathcal{Y}_n \bowtie \mathcal{X})$$

Proof. We prove the claim by showing set inclusion both ways. Let $z \in \mathsf{P}_f$. By expanding definitions, we get the following equivalences:

$$\begin{aligned}
& z \in \mathcal{X} \bowtie \left(\bigcup_{n\geq 0} \mathcal{Y}_n\right) \\
& \Leftrightarrow z \in \downarrow_{\sqsubseteq} \{x \bowtie y \mid x \in \mathcal{X} \wedge y \in \textstyle\bigcup_{n\geq 0} \mathcal{Y}_n\} && \{\text{Definition 3.3.5}\} \\
& \Leftrightarrow \exists x, y \in \mathsf{P}_f \colon x \in \mathcal{X} \wedge y \in \bigcup_{n\geq 0} \mathcal{Y}_n \wedge z \sqsubseteq x \bowtie y && \{\text{Definition of } \downarrow_{\sqsubseteq}\} \\
& \Leftrightarrow \exists n \geq 0 \colon \exists x, y \in \mathsf{P}_f \colon x \in \mathcal{X} \wedge y \in \mathcal{Y}_n \wedge z \sqsubseteq x \bowtie y && \{\text{Definition of set union}\} \\
& \Leftrightarrow \exists n \geq 0 \colon z \in \downarrow_{\sqsubseteq} \{x \bowtie y \mid x \in \mathcal{X} \wedge y \in \mathcal{Y}_n\} && \{\text{Definition of } \downarrow_{\sqsubseteq}\} \\
& \Leftrightarrow \exists n \geq 0 \colon z \in \mathcal{X} \bowtie \mathcal{Y}_n && \{\text{Definition 3.3.5}\} \\
& \Leftrightarrow z \in \bigcup_{n\geq 0} (\mathcal{X} \bowtie \mathcal{Y}_n) && \{\text{Definition of set union}\}
\end{aligned}$$

An analogous argument proves the second equation. □

As a corollary, it follows that the sequential and concurrent program composition operators are monotonic in both their arguments.

Corollary D.0.5 (Monotonicity)**.** *For every program $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$ in $\mathbb{P}$, if $\mathcal{X} \subseteq \mathcal{Y}$, then $\mathcal{X} \bowtie \mathcal{Z} \subseteq \mathcal{Y} \bowtie \mathcal{Z}$ and $\mathcal{Z} \bowtie \mathcal{X} \subseteq \mathcal{Z} \bowtie \mathcal{Y}$.*

Proof. Assume $\mathcal{X} \subseteq \mathcal{Y}$. Equivalently, $\mathcal{X} \cup \mathcal{Y} = \mathcal{Y}$. Thus, by Proposition D.0.4, $(\mathcal{Z} \bowtie \mathcal{X}) \cup (\mathcal{Z} \bowtie \mathcal{Y}) \subseteq \mathcal{Z} \bowtie (\mathcal{X} \cup \mathcal{Y}) = \mathcal{Z} \bowtie \mathcal{Y}$. Hence, $\mathcal{Z} \bowtie \mathcal{X} \subseteq \mathcal{Z} \bowtie \mathcal{Y}$, proving that concurrent and sequential composition are monotonic in their first argument. Similarly, for the second argument. □

By the Knaster-Tarski fixed point theorem and the fact that $\mathbb{P}$ is a complete lattice, it follows that 'iterative' concurrent and sequential composition of programs have a (necessarily unique) least fixed point solution. We denote this fixed point by two forms of Kleene star operators.

Proposition D.0.6. *Let $\mathcal{P}$ be a program and $F_{\mathcal{P}} \colon \mathbb{P} \to \mathbb{P}$ such that, for all $\mathcal{X} \in \mathbb{P}$, $F_{\mathcal{P}}(\mathcal{X}) = 1 \cup (\mathcal{P} \bowtie \mathcal{X})$. Then $F_{\mathcal{P}}$ has a least fixed point that we denote by $\mathcal{P}^{\bowtie}$.*

Proof. The conclusion follows from the fact that $\langle \mathbb{P}, \subseteq, \cap, \cup, 0 \rangle$ forms a complete lattice, Corollary D.0.5 and Knaster-Tarski fixed point theorem. □

Therefore, given any program $\mathcal{P}$ in $\mathbb{P}$ according to Definition 3.3.1, we have that $1 \cup (\mathcal{P}; \mathcal{P}^{;}) = \mathcal{P}^{;}$. In addition, for every program $\mathcal{Q}$ in $\mathbb{P}$, if $1 \cup (\mathcal{P}; \mathcal{Q}) = \mathcal{Q}$, then $\mathcal{P}^{;} \subseteq \mathcal{Q}$, and similarly for $\|$. By the Kleene fixed point theorem, $\mathcal{P}^{\|}$ and $\mathcal{P}^{;}$ for a program $\mathcal{P}$ can be computed as follows:

Proposition D.0.7. *Let $\mathcal{P}$ be a program in $\mathbb{P}$. Let $F_{\mathcal{P}} \colon \mathbb{P} \to \mathbb{P}$ be a function such that, for all programs $\mathcal{X} \in \mathbb{P}$, $F_{\mathcal{P}}(\mathcal{X}) = 1 \cup (\mathcal{P} \bowtie \mathcal{X})$. Then $\mathcal{P}^{\bowtie} = \bigcup_{n \geq 1} F_{\mathcal{P}}^{n}(0)$ where $0 \in \mathbb{P}$ and $F_{\mathcal{P}}^{1} \triangleq F$ and $F_{\mathcal{P}}^{j+1} \triangleq F_{\mathcal{P}} \circ F_{\mathcal{P}}^{j}$ for all $j \in \mathbb{N}$.*

Proof. By Proposition D.0.4, $F_{\mathcal{P}}$ is continuous. The conclusion follows from the Kleene fixed point theorem. □

The following result uses the transitivity of $\sqsubseteq$ (Proposition 3.2.15) to make clear the connection between partial string and program operators. This is key to transfer our knowledge about partial strings to programs.

Lemma D.0.8. *For all programs $\mathcal{U}, \mathcal{V}, \mathcal{X}, \mathcal{Y} \in \mathbb{P}$ and pairs of binary operators $\bowtie_a$ and $\bowtie_b$ in $\{;, \|\}$, if $\forall x, y \in \mathsf{P}_f \colon x \bowtie_a y \sqsubseteq x \bowtie_b y$, then $\mathcal{X} \bowtie_a \mathcal{Y} \subseteq \mathcal{X} \bowtie_b \mathcal{Y}$; similarly, if $\forall u, v, x, y \in \mathsf{P}_f \colon (u \bowtie_b v) \bowtie_a (x \bowtie_b y) \sqsubseteq (u \bowtie_a x) \bowtie_b (v \bowtie_a y)$ and $\bowtie_a$ is monotonic, then $(\mathcal{U} \bowtie_b \mathcal{V}) \bowtie_a (\mathcal{X} \bowtie_b \mathcal{Y}) \subseteq (\mathcal{U} \bowtie_a \mathcal{X}) \bowtie_b (\mathcal{V} \bowtie_a \mathcal{Y})$.*

Proof. Let $z \in \mathsf{P}_f$. Assume $z \in \mathcal{X} \bowtie_a \mathcal{Y}$. By Definition 3.3.5, $\mathcal{X} \bowtie_a \mathcal{Y} = {\downarrow_{\sqsubseteq}} \{x \bowtie_a y \mid x \in \mathcal{X} \wedge y \in \mathcal{Y}\}$. So there exists $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ such that $z \sqsubseteq x \bowtie_a y$. By hypothesis and transitivity of $\sqsubseteq$ (Proposition 3.2.15), there exists $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ such that $z \sqsubseteq x \bowtie_b y$. Thus $z \in \mathcal{X} \bowtie_b \mathcal{Y}$ because $\mathcal{X} \bowtie_b \mathcal{Y} = {\downarrow_{\sqsubseteq}} (\mathcal{X} \bowtie_b \mathcal{Y})$, i.e. $\mathcal{X} \bowtie_b \mathcal{Y}$ is a program. Since z is arbitrary, $\mathcal{X} \bowtie_a \mathcal{Y} \subseteq \mathcal{X} \bowtie_b \mathcal{Y}$.

The proof of the second implication is analogous except that it also uses the monotonicity of $\bowtie_a$ on partial strings. □

From the first implication of Lemma D.0.8 and Proposition 3.2.19 follows that the chain of composition operators carries over to $\langle \mathbb{P}, \subseteq \rangle$. The next proposition is the first of three frame laws [HvSM+14].

Proposition D.0.9 (Frame I)**.** *For all programs $\mathcal{X}, \mathcal{Y} \in \mathbb{P}$, $\mathcal{X}; \mathcal{Y} \subseteq \mathcal{X} \parallel \mathcal{Y}$.*

Proof. The conclusion follows from Proposition 3.2.19 and Lemma D.0.8. □

Noteworthy, the second implication of Lemma D.0.8, in turn, has as consequence that the exchange law for partial strings (Proposition 3.2.22) generalizes to program composition operators.

Proposition D.0.10. *For all* $\mathcal{U}, \mathcal{V}, \mathcal{X}, \mathcal{Y} \in \mathbb{P}$, $(\mathcal{U} \parallel \mathcal{V}); (\mathcal{X} \parallel \mathcal{Y}) \subseteq (\mathcal{U}; \mathcal{X}) \parallel (\mathcal{V}; \mathcal{Y})$.

Proof. By Proposition 3.2.22 and the second implication of Lemma D.0.8. □

Lemma D.0.11. *Let* $x, y, z, p \in \mathsf{P}_f$. *If* $x \bowtie y \cong y \bowtie x$, *then* $p \sqsubseteq x \bowtie y$ *is equivalent to* $p \sqsubseteq y \bowtie x$. *Similarly, if* $x \bowtie (y \bowtie z) \cong (x \bowtie y) \bowtie z$, *then* $p \sqsubseteq x \bowtie (y \bowtie z)$ *is equivalent to* $p \sqsubseteq (x \bowtie y) \bowtie z$.

Proof. The first hypothesis is $x \bowtie y \cong y \bowtie x$. Assume $p \sqsubseteq x \bowtie y$. By hypothesis and Proposition 3.2.17, $x \bowtie y \sqsubseteq y \bowtie x$. By transitivity of $\sqsubseteq$ (Proposition 3.2.15) and assumption, $p \sqsubseteq y \bowtie x$, proving the forward implication. The backward implication is proved analogously, as well as the second equivalence. □

Lemma D.0.12. *If* $\forall x, y \in \mathsf{P}_f \colon x \bowtie y \cong y \bowtie x$, *then* $\forall \mathcal{X}, \mathcal{Y} \in \mathbb{P} \colon \mathcal{X} \bowtie \mathcal{Y} = \mathcal{Y} \bowtie \mathcal{X}$. *Similarly, if* $\forall x, y \in \mathsf{P}_f \colon x \bowtie (y \bowtie z) \cong (x \bowtie y) \bowtie z$, *then* $\forall \mathcal{X}, \mathcal{Y}, \mathcal{Z} \in \mathbb{P} \colon \mathcal{X} \bowtie (\mathcal{Y} \bowtie \mathcal{Z}) = (\mathcal{X} \bowtie \mathcal{Y}) \bowtie \mathcal{Z}$.

Proof. Let $\mathcal{X}, \mathcal{Y} \in \mathbb{P}$ be programs and $p \in \mathsf{P}_f$ be a finite partial string. Assume $\forall x, y \in \mathsf{P}_f \colon x \bowtie y \cong y \bowtie x$. We show $\mathcal{X} \bowtie \mathcal{Y} \subseteq \mathcal{Y} \bowtie \mathcal{X}$ and $\mathcal{X} \bowtie \mathcal{Y} \supseteq \mathcal{Y} \bowtie \mathcal{X}$ through the following equivalences:

$$
\begin{aligned}
& p \in \mathcal{X} \bowtie \mathcal{Y} \\
& \Leftrightarrow p \in {\downarrow_{\sqsubseteq}} \{x \bowtie y \mid x \in \mathcal{X} \wedge y \in \mathcal{Y}\} && \{\text{Definition 3.3.5}\} \\
& \Leftrightarrow \exists x, y \in \mathsf{P}_f \colon x \in \mathcal{X} \wedge y \in \mathcal{Y} \wedge p \sqsubseteq x \bowtie y && \{\text{Definition of } {\downarrow_{\sqsubseteq}}\} \\
& \Leftrightarrow \exists x, y \in \mathsf{P}_f \colon x \in \mathcal{X} \wedge y \in \mathcal{Y} \wedge p \sqsubseteq y \bowtie x \\
& && \{\text{Assumption, commutativity of } \bowtie, \text{ Lemma D.0.11}\} \\
& \Leftrightarrow \exists x, y \in \mathsf{P}_f \colon y \in \mathcal{Y} \wedge x \in \mathcal{X} \wedge p \sqsubseteq y \bowtie x && \{\text{Commutativity of conjunction}\} \\
& \Leftrightarrow p \in {\downarrow_{\sqsubseteq}} \{y \bowtie x \mid y \in \mathcal{Y} \wedge x \in \mathcal{X}\} && \{\text{Definition of } {\downarrow_{\sqsubseteq}}\} \\
& \Leftrightarrow p \in \mathcal{Y} \bowtie \mathcal{X} && \{\text{Definition 3.3.5}\}
\end{aligned}
$$

Similarly, for associativity, it suffices to prove subset inclusion both ways which we show in detail to draw attention to the various properties of $\sqsubseteq$ and $\bowtie$ that are

used in the proof:

$$
\begin{aligned}
& p \in \mathcal{X} \bowtie (\mathcal{Y} \bowtie \mathcal{Z}) \\
& \Leftrightarrow p \in \downarrow_{\sqsubseteq} \{x \bowtie q \mid x \in \mathcal{X} \wedge q \in (\mathcal{Y} \bowtie \mathcal{Z})\} && \{\text{Definition 3.3.5}\} \\
& \Leftrightarrow \exists x, q \in \mathsf{P}_f \colon x \in \mathcal{X} \wedge q \in (\mathcal{Y} \bowtie \mathcal{Z}) \wedge p \sqsubseteq x \bowtie q && \{\text{Definition of } \downarrow_{\sqsubseteq}\} \\
& \Leftrightarrow \exists x, q \in \mathsf{P}_f \colon x \in \mathcal{X} \wedge q \in \downarrow_{\sqsubseteq} \{y \bowtie z \mid y \in \mathcal{Y} \wedge z \in \mathcal{Z}\} \wedge p \sqsubseteq x \bowtie q \\
& && \{\text{Definition 3.3.5}\} \\
& \Leftrightarrow \exists x, q \in \mathsf{P}_f \colon x \in \mathcal{X} \wedge (\exists y, z \in \mathsf{P}_f \colon y \in \mathcal{Y} \wedge z \in \mathcal{Z} \wedge q \sqsubseteq y \bowtie z) \wedge p \sqsubseteq x \bowtie q \\
& && \{\text{Definition of } \downarrow_{\sqsubseteq}\} \\
& \Leftrightarrow \exists x, y, z, q \in \mathsf{P}_f \colon x \in \mathcal{X} \wedge y \in \mathcal{Y} \wedge z \in \mathcal{Z} \wedge q \sqsubseteq y \bowtie z \wedge p \sqsubseteq x \bowtie q \\
& && \{\text{Definition of } \exists\} \\
& \Rightarrow \exists x, y, z, q \in \mathsf{P}_f \colon x \in \mathcal{X} \wedge y \in \mathcal{Y} \wedge z \in \mathcal{Z} \wedge x \bowtie q \sqsubseteq x \bowtie (y \bowtie z) \wedge p \sqsubseteq x \bowtie q \\
& && \{\text{Monotonicity of } \bowtie\} \\
& \Rightarrow \exists x, y, z \in \mathsf{P}_f \colon x \in \mathcal{X} \wedge y \in \mathcal{Y} \wedge z \in \mathcal{Z} \wedge p \sqsubseteq x \bowtie (y \bowtie z) && \{\text{transitivity of } \sqsubseteq\} \\
& \Leftrightarrow \exists x, y, z \in \mathsf{P}_f \colon x \in \mathcal{X} \wedge y \in \mathcal{Y} \wedge z \in \mathcal{Z} \wedge p \sqsubseteq (x \bowtie y) \bowtie z \\
& && \{\text{Assumption, associativity of } \bowtie, \text{ Lemma D.0.11}\} \\
& \Rightarrow \exists x, y, z, q' \in \mathsf{P}_f \colon x \in \mathcal{X} \wedge y \in \mathcal{Y} \wedge z \in \mathcal{Z} \wedge q' \sqsubseteq x \bowtie y \wedge p \sqsubseteq q' \bowtie z \\
& && \{\text{Reflexivity and transitivity of } \sqsubseteq, \text{ monotonicity of } \bowtie\} \\
& \Leftrightarrow p \in (\mathcal{X} \bowtie \mathcal{Y}) \bowtie \mathcal{Z} && \{\text{Definition 3.3.5}\}
\end{aligned}
$$

An analogous argument proves $\mathcal{X} \bowtie (\mathcal{Y} \bowtie \mathcal{Z}) \supseteq (\mathcal{X} \bowtie \mathcal{Y}) \bowtie \mathcal{Z}$, proving associativity of the $\bowtie$ operator on programs. □

We remark that the previous lemma has as antecedent that a partial string operator is commutative (or associative) only up to isomorphism. In contrast, the consequent of the lemma asserts that programs are in fact *equal* when composed with the corresponding program composition operator. This equality is due to the fact that programs are downward-closed with respect to an ordering which disregards the identity of events.

Definition D.0.13. A **semigroup** is an algebraic structure consisting of a set together with an associative binary operation.

Proposition D.0.14. *$\langle \mathbb{P}, \| \rangle$ is a commutative semigroup and $\langle \mathbb{P}, ; \rangle$ is a semigroup.*

Proof. Let $\mathcal{X}, \mathcal{Y}, \mathcal{Z} \in \mathbb{P}$ be programs. By modus ponens, Lemma D.0.12 and Proposition 3.2.21, $\mathcal{X} \bowtie (\mathcal{Y} \bowtie \mathcal{Z}) = (\mathcal{X} \bowtie \mathcal{Y}) \bowtie \mathcal{Z}$. Similarly, $\mathcal{X} \parallel \mathcal{Y} = \mathcal{Y} \parallel \mathcal{X}$. □

The fact that $\|$ is a commutative program operator has the following weak principle of sequential consistency as consequence [HvSM+14]:

Proposition D.0.15. *For all programs* $\mathcal{X}, \mathcal{Y} \in \mathbb{P}$, $(\mathcal{X}; \mathcal{Y}) \cup (\mathcal{Y}; \mathcal{X}) \subseteq \mathcal{X} \parallel \mathcal{Y}$.

Proof. By Proposition D.0.9 and D.0.14, $(\mathcal{X}; \mathcal{Y}) \subseteq \mathcal{X} \parallel \mathcal{Y}$ and $(\mathcal{Y}; \mathcal{X}) \subseteq \mathcal{X} \parallel \mathcal{Y}$. Thus, by definition of least upper bound, $(\mathcal{X}; \mathcal{Y}) \cup (\mathcal{Y}; \mathcal{X}) \subseteq \mathcal{X} \parallel \mathcal{Y}$. □

The converse of the previous proposition does not generally hold, a good examplar of the fact that a set of partial strings is more expressive than a set of strings. This link to classical language theory is formalized as follows:

Definition D.0.16. A **string** is a finite partial string s in P_f such that $\preceq_s$ is a total order, i.e. for all $e, e' \in E_s$, $e \preceq_s e'$ or $e' \preceq_s e$. Let Γ^* be the set of strings. For all programs $\mathcal{P}$ in $\mathbb{P}$, define the **language of** $\mathcal{P}$, written $\mathfrak{L}_{\mathcal{P}}$, to be the set of strings where each one refines at least one partial string in $\mathcal{P}$; equivalently, $\mathfrak{L}_{\mathcal{P}} \triangleq \{s \in \Gamma^* \mid \exists p \in \mathcal{P} \colon s \sqsubseteq p\}$.

In other words, $\mathfrak{L}_{\mathcal{P}}$ can be seen as the linearizations of all the partial strings in a program according to the refinement ordering of Definition 3.2.6.

Proposition D.0.17. *For all programs* $\mathcal{X}, \mathcal{Y} \in \mathbb{P}$, *if* $\mathcal{X} \subseteq \mathcal{Y}$, *then* $\mathfrak{L}_{\mathcal{X}} \subseteq \mathfrak{L}_{\mathcal{Y}}$.

Proof. Immediate from Definition D.0.16. □

The converse of Proposition D.0.17 does not generally hold. For example, $\mathfrak{L}_{\mathcal{X} \parallel \mathcal{Y}} \subseteq \mathfrak{L}_{(\mathcal{X};\mathcal{Y}) \cup (\mathcal{Y};\mathcal{X})}$ but $\mathcal{X} \parallel \mathcal{Y} \not\subseteq (\mathcal{X}; \mathcal{Y}) \cup (\mathcal{Y}; \mathcal{X})$ for some programs $\mathcal{X}$ and $\mathcal{Y}$. This shows that our notion of programs from Definition 3.3.1 strictly generalizes the concept of sets of strings, i.e. languages consisting of strings.

Definition D.0.18. A **monoid** is a semigroup with an identity element.

Proposition D.0.19. $\langle \mathbb{P}, \|, 1 \rangle$ *is a commutative monoid and* $\langle \mathbb{P}, ;, 1 \rangle$ *is a monoid.*

Proof. Let $\mathcal{X} \in \mathbb{P}$ be a program. By Proposition D.0.14, it remains to show that 1 is the identity for the operator $\bowtie$ on programs, as shown in Proposition D.0.3. □

Of particular interest are complete lattices under the natural order (formally, $x \leq y \triangleq x \vee y = y$ where $\vee$ denotes join) in which a binary operator distributes over arbitrary least upper bounds ($\bigvee$).

Definition D.0.20. A **quantale** is a complete lattice Q equipped with a semigroup structure $\langle Q, \cdot \rangle$ satisfying both complete distributive laws

$$x \cdot \left(\bigvee S \right) = \bigvee \{ y \cdot x \mid y \in S \}$$
$$\left(\bigvee S \right) \cdot x = \bigvee \{ x \cdot y \mid y \in S \}$$

where $S \subseteq Q$ and $x \in Q$. If Q is a quantale whose semigroup is also a monoid structure $\langle Q, \cdot, 1 \rangle$, then Q is called a **unital quantale**.

Quantales appear in different guises in computer science. For example, the power-set of strings over alphabet Γ is a unital quantale $\langle \mathcal{P}(\Gamma^*), \subseteq, \cup, 0, 1, \cdot \rangle$ where $\subseteq$ is the inclusion order on sets, $\cup$ is set union, $0 \triangleq \emptyset$ is the empty set, $1 \triangleq \{\varepsilon\}$ is the singleton set with the empty string ε, and the concatenation of two sets of strings, A and B, is defined pair-wise, i.e. $A \cdot B \triangleq \{a \cdot b \mid a \in A \wedge b \in B\}$, whence $\langle \mathcal{P}(\Gamma^*), 1, \cdot \rangle$ forms a monoid. Based on the notion of quantales, we succinctly characterize the complete lattice of programs as follows:

Proposition D.0.21. *The algebraic structure $\langle \mathbb{P}, \subseteq, \cup, 0, 1, ;, \| \rangle$ consists of two unital quantales with respect to sequential and concurrent program composition operators, respectively, that together satisfy the exchange law (Proposition D.0.10).*

Proof. By Proposition D.0.4 and D.0.19, $\langle \mathbb{P}, \subseteq, \| \rangle$ and $\langle \mathbb{P}, \subseteq, ; \rangle$ form two quantales according to Definition D.0.20. □

Proposition D.0.22. *For all $\mathcal{X}, \mathcal{Y}, \mathcal{Z} \in \mathbb{P}$, the following holds:*

$$\mathcal{X} \cup (\mathcal{Y} \cup \mathcal{Z}) = (\mathcal{X} \cup \mathcal{Y}) \cup \mathcal{Z}$$
$$\mathcal{X} \cup \mathcal{Y} = \mathcal{Y} \cup \mathcal{X}$$
$$\mathcal{X} \cup \mathcal{X} = \mathcal{X}$$
$$\mathcal{X} \cup 0 = 0 \cup \mathcal{X} = 0$$

Proof. All these equalities are true since $\cup$ is the least upper bound in the complete lattice $\mathbb{P}$ whose bottom element is 0. □

The next proposition shows that the set of programs forms an algebraic structure commonly known as either an idempotent semiring or dioid. A familiar example of an idempotent semiring is the Boolean semiring $\langle \mathbb{B}, +, \cdot, 0, 1 \rangle$ where $\mathbb{B}$ should be interpreted as Boolean values whose sum and product operators are logical disjunction and conjunction, respectively.

Proposition D.0.23. *The algebraic structures* $\langle \mathbb{P}, \cup, \|, 0, 1 \rangle$ *and* $\langle \mathbb{P}, \cup, ;, 0, 1 \rangle$ *are idempotent semirings where* $\langle \mathbb{P}, \cup, 1 \rangle$ *is a commutative idempotent monoid,* $\langle \mathbb{P}, \|, 1 \rangle$ *is a commutative monoid and* $\langle \mathbb{P}, ;, 1 \rangle$ *is a monoid.*

Proof. By Proposition D.0.2, D.0.19 and D.0.22. □

Finally, Theorem 3.3.6 follows from Proposition D.0.7, D.0.10, D.0.23 and D.0.21.

Appendix E

Patch for quadratic-size partial-order encoding in CBMC

This appendix gives the most essential implementation details for the quadratic-size partial-order encoding from Chapter 4. In particular, the patch in Listing E.1 shows how to implement the quadratic-size SAT-based partial-order encoding in CBMC 5.2.

Our patch relies on the pure virtual `partial_order_concurrencyt::before(event_it, event_it)` function. We assume that it symbolically encodes an antisymmetric binary relation. This assumption holds under CBMC 5.2's bit-wise encoding. Moreover, we assume that each `event` iterator is stable. This assumption is warranted since events are stored in a `std::list`.

The patch introduces two new member fields into the `partial_order_concurrencyt` class: `sup_event_it` and `sup_events`. Our design assumes that each read event can be essentially renamed to a unique write event by simply changing the event's type from `goto_trace_stept::SHARED_READ` to `goto_trace_stept::SHARED_WRITE`. This is how the procedure `partial_order_concurrencyt::create_sup_event(event_it)` creates a unique supremum event for a given read event. The newly created supremum event is then stored in `sup_events`. In doing so, the `sup_event_it` iterator is updated to point to the end of `sup_events`. This avoids having to call `std::prev` on `sup_events.end()` which would require the returned iterator to be bi-directional.[1] Instead, we call the increment operator on `sup_event_it` whose semantics is defined by the simpler `ForwardIterator` concept.[2]

The implementation of the partial-order constraints works as follows. Foremost, the `ensure_supremum` constraint, for a read event `r`, defines `sup` to be an upper bound of $\mathcal{H}(\texttt{r})$ where $\mathcal{H}(\texttt{r})$ is a set that contains write events `w` on the same memory location

[1] http://en.cppreference.com/w/cpp/iterator/prev
[2] http://en.cppreference.com/w/cpp/concept/ForwardIterator

as r such that not_exprt(before(r, w)), denoted by w <= r. We have modified the read_from constraint to look as follows:

```
implies_exprt read_from(s,
    and_exprt(w->guard, not_exprt(before(w, sup)),
      equal_exprt(r->ssa_lhs, w->ssa_lhs)));
```

Suppose s holds, i.e. r reads-from w (necessarily on the same memory location). By the rf-order constraint, we get before(w, r), which implies not_exprt(before(r, w)), i.e. w <= r. Thus w is in $\mathcal{H}(r)$. Since sup is an upper bound of $\mathcal{H}(r)$, it follows w <= sup. By the new read_from constraint, sup <= w. Since we assume that partial_order_concurrencyt::before(event_it, event_it) is antisymmetric, we conclude that the clocks of sup and w are equal. This is what we needed to implement strong read consistency. If we put this all together, here is what the entire patch looks like:

```
Index: src/goto-symex/memory_model.cpp
===================================================================
--- src/goto-symex/memory_model.cpp (revision 5834)
+++ src/goto-symex/memory_model.cpp (working copy)
@@ -120,10 +120,19 @@
          r_it++)
     {
       const event_it r=*r_it;

       exprt::operandst rf_some_operands;
       rf_some_operands.reserve(a_rec.writes.size());

+#ifdef CPROVER_MEMORY_MODEL_SUP
+       // To implement the quadratic-size partial order encoding, we
+       // create a formula that finds the least upper bound (supremum)
+       // of all writes that do not happen after a read on the same
+       // memory location. Since there are different memory axioms,
+       // we create a new event rather than a new clock variable.
+       const event_it sup=create_sup_event(r);
+#endif
+
       // this is quadratic in #events per address
       for(event_listt::const_iterator
           w_it=a_rec.writes.begin();
@@ -140,6 +149,15 @@
           w->source.thread_nr==r->source.thread_nr;

         symbol_exprt s=nondet_bool_symbol("rf");
+
+#ifdef CPROVER_MEMORY_MODEL_SUP
+        implies_exprt ensure_supremum(
+          and_exprt(w->guard, not_exprt(before(r, w))),
+          not_exprt(before(sup, w)));
+
+        add_constraint(equation,
+          ensure_supremum, "sup-clock", r->source);
+#endif
```

```
         // record the symbol
         choice_symbols[
@@ -149,6 +167,9 @@
         // one write event that has guard 'true'.
         implies_exprt read_from(s,
             and_exprt(w->guard,
+#ifdef CPROVER_MEMORY_MODEL_SUP
+                       not_exprt(before(w, sup)),
+#endif
               equal_exprt(r->ssa_lhs, w->ssa_lhs)));

         // Uses only the write's guard as precondition, read's guard
Index: src/goto-symex/memory_model_pso.cpp
===================================================================
--- src/goto-symex/memory_model_pso.cpp (revision 5834)
+++ src/goto-symex/memory_model_pso.cpp (working copy)
@@ -30,7 +30,7 @@
   read_from(equation);
   write_serialization_external(equation);
   program_order(equation);
-#ifndef CPROVER_MEMORY_MODEL_SUP_CLOCK
+#ifndef CPROVER_MEMORY_MODEL_SUP
   from_read(equation);
 #endif
 }
Index: src/goto-symex/memory_model_sc.cpp
===================================================================
--- src/goto-symex/memory_model_sc.cpp  (revision 5834)
+++ src/goto-symex/memory_model_sc.cpp  (working copy)
@@ -33,7 +33,9 @@
   read_from(equation);
   write_serialization_external(equation);
   program_order(equation);
+#ifndef CPROVER_MEMORY_MODEL_SUP
   from_read(equation);
+#endif
 }

 /*******************************************************************\
Index: src/goto-symex/memory_model_tso.cpp
===================================================================
--- src/goto-symex/memory_model_tso.cpp (revision 5834)
+++ src/goto-symex/memory_model_tso.cpp (working copy)
@@ -33,7 +33,7 @@
   read_from(equation);
   write_serialization_external(equation);
   program_order(equation);
-#ifndef CPROVER_MEMORY_MODEL_SUP_CLOCK
+#ifndef CPROVER_MEMORY_MODEL_SUP
   from_read(equation);
 #endif
 }
Index: src/goto-symex/partial_order_concurrency.cpp
===================================================================
```

```
--- src/goto-symex/partial_order_concurrency.cpp  (revision 5834)
+++ src/goto-symex/partial_order_concurrency.cpp  (working copy)
@@ -27,7 +27,9 @@
 \*****************************************************************/

 partial_order_concurrencyt::partial_order_concurrencyt(
-  const namespacet &_ns):ns(_ns)
+  const namespacet &_ns):ns(_ns),
+  sup_events(),
+  sup_event_it(sup_events.end())
 {
 }

@@ -47,8 +49,45 @@
 {
 }

+#ifdef CPROVER_MEMORY_MODEL_SUP
 /*****************************************************************\

+Function: partial_order_concurrencyt::create_sup_event
+
+  Inputs: Read event, denoted by r
+
+ Outputs: A hidden write event on the same memory location as r
+
+ Purpose: Find the supremum of writes that do not happen after r
+
+\*****************************************************************/
+
+partial_order_concurrencyt::event_it
+  partial_order_concurrencyt::create_sup_event(event_it r)
+{
+  assert(is_shared_read(r));
+
+  sup_events.push_back(eventt());
+  eventt &sup_event=sup_events.back();
+
+  if(sup_event_it==sup_events.end())
+    sup_event_it=sup_events.begin();
+  else
+    ++sup_event_it;
+
+  sup_event.guard=false_exprt();
+  sup_event.ssa_lhs=r->ssa_lhs;
+  sup_event.original_lhs_object=r->original_lhs_object;
+  sup_event.type=goto_trace_stept::SHARED_WRITE;
+  sup_event.atomic_section_id=r->atomic_section_id;
+  sup_event.source=r->source;
+
+  return sup_event_it;
+}
+#endif
+
+/*****************************************************************\
```

```
+
 Function: partial_order_concurrencyt::add_init_writes

   Inputs:
Index: src/goto-symex/partial_order_concurrency.h
===================================================================
--- src/goto-symex/partial_order_concurrency.h  (revision 5834)
+++ src/goto-symex/partial_order_concurrency.h  (working copy)
@@ -13,6 +13,8 @@

 #include "symex_target_equation.h"

+#define CPROVER_MEMORY_MODEL_SUP
+
 class partial_order_concurrencyt:public messaget
 {
 public:
@@ -38,6 +40,9 @@
 protected:
   const namespacet &ns;

+  eventst sup_events;
+  event_it sup_event_it;
+
   typedef std::vector<event_it> event_listt;

   // lists of reads and writes per address
@@ -73,6 +78,9 @@
   symbol_exprt clock(event_it e, axiomt axiom);
   void build_clock_type(const symex_target_equationt &);

+  // returns a new supremum event for a read event r
+  event_it create_sup_event(event_it r);
+
   // preprocess and add a constraint to equation
   void add_constraint(
     symex_target_equationt &equation,
```

Listing E.1: CBMC 5.2 patch for the quadratic-size partial-order encoding.

Appendix F

Detailed OpenCores Ethernet MAC benchmark results

	Glucose 4.0							
	Time (*min*)			Max memory accesses			CNF Formula	
Property	Total	SAT	Memory (*MB*)	# reads	# writes	# SSA steps	# variables	# clauses
(HW.1)	1.4	1.4	760.6	8	228	6,605	804,534	4,078,743
(HW.2)	125.2	125.0	7317.4	79	233	60,177	7,706,109	51,292,186
(HW.3)	16.4	16.3	5562.5	74	234	59,305	7,772,958	49,648,078
(HW.4)	62.5	62.3	5937.1	78	234	61,353	7,915,065	51,868,959
(HW.5)	101.4	101.1	8458.1	108	236	87,261	10,785,871	74,541,135
(HW.1–5)	14.5	N/A	MEMOUT	356	250	459,673	N/A	N/A
	MiniSAT 2.2							
(HW.1)	1.5	1.5	599.8	8	228	6,605	804,534	4,078,743
(HW.2)	22.3	22.1	6596.0	79	233	60,177	7,706,109	51,292,186
(HW.3)	11.9	11.7	6372.2	74	234	59,305	7,772,958	49,648,078
(HW.4)	22.4	22.2	6616.8	78	234	61,353	7,915,065	51,868,959
(HW.5)	35.7	35.4	9356.9	108	236	87,261	10,785,871	74,541,135
(HW.1–5)	19.5	N/A	MEMOUT	356	250	459,673	N/A	N/A

Table F.1: Experimental results for the OpenCores Ethernet MAC benchmark using our CBMC implementation of the quadratic-size partial-order encoding.

	Cubic-size encoding				Quadratic-size encoding			
	Glucose 4.0							
Property	Starts	Decisions	Conflicts	Learnt Literals	Starts	Decisions	Conflicts	Learnt Literals
(HW.1)	8	10,280	2,562	18,721	1,220	1,623,392	222,722	1,007,463
(HW.2)	79	6,872,030	51,553	1,357,578	1,347	131,332,884	869,198	8,039,961
(HW.3)	171	10,652,762	27,931	575,904	592	228,906,094	253,617	3,304,871
(HW.4)	247	39,859,596	50,390	1,434,225	1,814	286,177,587	945,974	2,350,128
(HW.5)	141	9,271,375	80,981	6,606,360	1,494	336,186,099	601,686	15,885,726
	MiniSAT 2.2							
(HW.1)	14	8,318	2,120	31,188	492	2,242,710	177,981	13,539,126
(HW.2)	253	1,199,822	82,968	11,283,238	1,020	8,133,470	421,867	15,362,529
(HW.3)	125	881,303	34,855	1,118,330	377	6,478,649	135,054	5,954,947
(HW.4)	254	1,410,639	88,870	6,187,649	874	13,763,625	362,697	15,974,594
(HW.5)	318	1,454,880	120,490	25,366,977	959	15,890,231	397,614	24,549,522

Table F.2: SAT solver statistics for OpenCores Ethernet MAC benchmark.

This appendix compares in detail the cubic-size and quadratic-size partial-order encoding (Chapter 4) using the OpenCores Ethernet MAC benchmark (Chapter 2). We analyzed the experimental results for this comparison in Section 4.4.3.

	Glucose 4.0				
Property	SAT time (*min*)	Starts	Decisions	Conflicts	Learnt Literals
(HW.1)	1.1	968	2,048,650	173,555	865,903
(HW.2)	69.3	1096	60,158,831	559,111	2,787,970
(HW.3)	12.6	241	431,016,030	68,426	3,695,445
(HW.4)	64.9	1602	114,172,549	447,701	7,308,447
(HW.5)	54.1	1209	34,344,657	475,172	6,311,745
	MiniSAT 2.2				
(HW.1)	1.2	439	3,528,566	160,324	16,561,008
(HW.2)	19.1	854	6,155,295	357,217	20,428,860
(HW.3)	9.5	158	1,637,266	52,568	1,813,387
(HW.4)	23.7	766	7,352,085	321,327	30,664,533
(HW.5)	16.0	756	4,032,656	307,488	18,687,924

Table F.3: SAT solver statistics for OpenCores Ethernet MAC benchmark where supremum variables are translated to non-decision variables.

Appendix G

Details on concurrency challenge problem

This appendix contains more detailed proofs (Section G.1) and experimental results (Section G.2) concerning the concurrency challenge problem from Chapter 5.

G.1 Proofs for lower bounds

This section gives a more formal derivation for ν_π and M_π the concepts discussed in Section 5.5, and proofs for Lemma 5.5.1 and Theorem 5.5.3. We use $\mu, \nu \models \phi$ to denote that a Σ-structure μ and a variable assignment ν over Σ-structure μ satisfies a Σ-formula ϕ.

Let μ be any $(\mathcal{T}_C + \mathcal{T}_S + \mathcal{T}_V)$-structure with the additional constraint that $\mathcal{T}_C$, $\mathcal{T}_S$, and $\mathcal{T}_V$ sorts are mapped to domains with cardinalities at least $|E|$, $N+1$, and $N+1$ respectively. Such a structure exists unless $\mathcal{T}_V$ is bit-vectors and the bit-width is insufficiently large. We now construct a first-order variable assignment ν_π over $\mathcal{T}_C$ and $\mathcal{T}_S$ variables to match $\sigma(\pi)$. Let $x_1 \prec^\mu \cdots \prec^\mu x_{|E|}$ be any arbitrary chain in the $\mathcal{T}_C$ domain of μ, and let $\langle y_0, y_1, \ldots, y_N \rangle$ be an arbitrary enumeration of $N+1$ distinct elements in the $\mathcal{T}_S$ domain. Both the x_i chain and the y_i sequence exist as

the cardinalities are large enough. We assign the c_e and s_e variables as follows:

$$\nu_\pi(\mathsf{c}_e) = \begin{cases} x_1 & e = w_{init} \\ x_{2i} & e = r_{\pi(i)} \\ x_{2i+1} & e = w_{\pi(i)} \\ x_{2N+2} & e = r_{assert} \end{cases}$$

$$\nu_\pi(\mathsf{s}_w) = \begin{cases} y_0 & w = w_{init} \\ y_i & w = w_{\pi(i)} \end{cases}$$

$$\nu_\pi(\mathsf{s}_r) = \begin{cases} y_i & r = r_{\pi(i+1)} \\ y_N & r = r_{assert} \end{cases}$$

We construct a complete set of $\mathcal{T}$-literals H_π (either $\ell \in H_\pi$ or $\neg\ell \in H_\pi$ for all $\ell \in \mathcal{L}_\mathcal{A}$). This will correspond to M_π before abstraction. For any literal ℓ over $\mathcal{T}_C$ or $\mathcal{T}_S$ atoms, we evaluate ℓ w.r.t. μ and ν_π to assign it in H_π, i.e. $\ell \in H_\pi$ if $\mu, \nu_\pi \models \ell$. For atoms over $\mathcal{T}_V$, we include the $\neg L_\pi$ literals in H_π (defined in Section 5.5). For all other $\mathcal{T}_V$ equalities ℓ in ϕ^3, we include $\neg\ell \in H_\pi$. We now let $M_\pi = {H_\pi}^\mathbb{B}$.

Proof of Lemma 5.5.1. Since $\neg L_\pi \subseteq H_\pi$ and $M_\pi = {H_\pi}^\mathbb{B}$, $\neg L_\pi^\mathbb{B} \subseteq M_\pi$. We now show that for each $\ell \in \neg L_\pi$, we can extend ν_π to a new assignment ν_π^ℓ so that $\mu, \nu_\pi^\ell \models h$ for all $h \in H_\pi \setminus \{\ell\}$. For brevity, we denote by $\ell_0 = \left(\mathsf{rv}_{r_{\pi(1)}} = 0\right)$, $\ell_i = \left(\mathsf{rv}_{r_{\pi(i)}} + 1 = \mathsf{rv}_{r_{\pi(i+1)}}\right)$ for $i \in 1 \ldots N-1$, $\ell_{assert1} = \left(\mathsf{rv}_{r_{\pi(N)}} + 1 = \mathsf{rv}_{r_{assert}}\right)$, and $\ell_{assert2} = (\mathsf{rv}_{r_{assert}} > N)$.

$$\nu_\pi^{\ell_0}(\mathsf{rv}_r) = \begin{cases} 1 & r = r_{\pi(1)} \\ j+1 & r = r_{\pi(j)} \\ N+1 & r = r_{assert} \end{cases} \qquad \nu_\pi^{\ell_i}(\mathsf{rv}_r) = \begin{cases} 0 & r = r_{\pi(1)} \\ j & r = r_{\pi(j)}, j < i \\ k+1 & r = r_{\pi(k)}, k \geq i \\ N+1 & r = r_{assert} \end{cases}$$

$$\nu_\pi^{\ell_{assert1}}(\mathsf{rv}_r) = \begin{cases} 0 & r = r_{\pi(1)} \\ j & r = r_{\pi(j)} \\ N+1 & r = r_{assert} \end{cases} \qquad \nu_\pi^{\ell_{assert2}}(\mathsf{rv}_r) = \begin{cases} 0 & r = r_{\pi(1)} \\ j & r = r_{\pi(j)} \\ N & r = r_{assert} \end{cases}$$

We omit $\_^\mu$ from the $\mathcal{T}_V$-constants $0, \ldots, N+1$ above. It is now that case that $\mu, \nu_\pi^\ell \models h$ for all $h \in H_\pi \setminus \{\ell\}$. Thus $H_\pi \setminus \{\ell\}$ is satisfiable modulo $\mathcal{T}$. As every subset

of H_π excluding exactly one literal in $\neg L_\pi$ is satisfiable modulo $\mathcal{T}$, $\neg L_\pi$ is the unique minimal $\mathcal{T}$-conflict in H_π. Thus M_π is a critical assignment. □

Proof of Theorem 5.5.3. We extend ν_π to assign sup_r to match $\sigma(\pi)$: $\nu_\pi(\mathsf{sup}_{r_{\pi(1)}}) = \nu_\pi(\mathsf{c}_{w_{init}})$, $\nu_\pi(\mathsf{sup}_{r_{\pi(i)}}) = \nu_\pi(\mathsf{c}_{w_{\pi(i-1)}})$, and $\nu_\pi(\mathsf{sup}_{r_{assert}}) = \nu_\pi(\mathsf{c}_{w_{\pi(N)}})$. We follow the same construction of H_π, M_π, ν_π^ℓ, and Q as before for ϕ^3. Q is a set of non-interfering critical assignments for ϕ^2. □

G.2 Experimental results

Tables G.1, G.3 and G.4 give the detailed experimental results plotted in Figure 5.3. Note that there is no standard that governs the reporting of SMT solver statistics. For example, Yices2 reports only 'theory conflicts' leaving implicit the fact that it is the theory of linear integer arithmetic.

	$\mathcal{E}^3$						$\mathcal{E}^2$					
	Conflicts			Lemmas	Propagations		Conflicts			Lemmas	Propagations	
N	SAT	BV	Arith	Arith	BV	Arith	SAT	BV	Arith	Arith	BV	Arith
cvc4-real-clocks-bv-val												
3	15	18	2	80	157	67	14	17	7	64	145	147
4	68	36	3	137	652	549	54	44	15	112	647	808
5	552	171	15	217	5155	4320	444	214	86	189	5155	6754
6	5122	925	44	322	44879	42282	4293	1266	271	292	45809	61441
7	53848	6022	136	455	446890	440396	52279	7881	494	420	454381	631589
8	642146	50783	287	619	5178188	5136835	1099060	60183	1088	578	6871070	10564897
cvc4-bv-clocks-int-val												
3	10	0	14	11	72	48	10	6	16	11	141	37
4	69	0	57	25	332	220	66	26	59	25	737	271
5	549	0	308	40	2478	1585	565	117	256	40	5314	1932
6	4278	0	1698	57	19073	10168	5086	434	1488	57	39796	11335
7	33929	0	11033	77	158325	48871	60243	908	11627	77	418431	51496
8	313224	0	85166	100	1388661	267599	1083503	1880	100512	100	6532492	295647
9	3514133	0	804965	126	15353222	1954639	N/A	N/A	N/A	N/A	N/A	N/A
z3-real-clocks-bv-val												
3	156	14	1	N/A	22694		125	10	1	N/A	15924	
4	670	26	N/A	N/A	129779		654	83	1	N/A	150867	
5	3349	44	5	N/A	749094		4026	127	6	N/A	1058745	
6	38078	750	1	N/A	9803060		40705	345	16	N/A	11083196	
7	652099	869	19	N/A	171387270		499290	604	20	N/A	145715448	
8	5568602	1406	33	N/A	1697831872		8848732	891	36	N/A	2847939426	
z3-bv-clocks-int-val												
3	133	N/A	11	N/A	27646		22	N/A	17	N/A	1636	
4	221	N/A	34	N/A	49320		113	N/A	45	N/A	7644	
5	972	N/A	241	N/A	150142		889	N/A	222	N/A	76612	
6	8027	N/A	2045	N/A	771377		6782	N/A	1511	N/A	584859	
7	45562	N/A	10182	N/A	4340112		42919	6	10986	N/A	3847401	
8	546303	6	149677	N/A	54348028		408597	9	108788	N/A	42353211	
9	5417266	256	1619484	N/A	581120650		4787134	75	1551724	N/A	589653075	

Table G.1: Concurrency challenge benchmark results for mixed BV/arithmetic.

	$\mathcal{E}^3$			$\mathcal{E}^2$		
N	Conflicts	Decisions	Propagations	Conflicts	Decisions	Propagations
	cvc4-bv-clocks-bv-val					
3	114	643	20178	74	403	20262
4	507	1614	120432	517	2069	125623
5	4988	18369	1365593	3957	11272	1283997
6	38318	124950	14142728	39694	130490	14878934
7	376556	1043697	154203853	354572	879648	155906988
	z3-bv-clocks-bv-val					
3	179	589	1753	169	668	1095
4	574	1728	8988	852	2794	9162
5	3838	9000	43494	5336	15602	67949
6	42420	90209	1103391	40730	81336	1064427
7	523734	1030036	26693996	628701	1123423	31424804
8	12115679	20732697	849617378	N/A	N/A	N/A
	yices2-bv-clocks-bv-val					
3	236	1270	N/A	236	1268	N/A
4	519	2179	N/A	519	2181	N/A
5	5997	15397	N/A	5997	15405	N/A
6	79205	192775	N/A	79205	192791	N/A
7	478620	1155931	N/A	478620	1155957	N/A
	boolector-bv-clocks-bv-val					
3	57	312	16840	56	279	17399
4	486	2061	149579	468	1806	164432
5	5193	21257	1076955	5749	21988	1133190
6	71045	228998	8866207	59344	212452	7755366
7	859460	2607824	91616678	823070	2639413	89748245
8	8246613	22615059	702660158	6647339	18622802	656629672
9	16986741	44150711	1516400694	16226921	41835641	1631307287

Table G.4: Concurrency challenge benchmark results for pure bit-vectors (`QF_BV`).

CVC4			Z3			Yices and Boolector		
N	Time (s)	Memory (MB)	N	Time (s)	Memory (MB)	N	Time (s)	Memory (MB)
	cvc4-real-clocks-int-val-$\mathcal{E}^3$			*z3-real-clocks-int-val-$\mathcal{E}^3$*			*yices-real-clocks-int-val-$\mathcal{E}^3$*	
3	0.00	0.0	3	0.00	0.0	3	0.00	0.0
4	0.00	0.0	4	0.00	0.0	4	0.00	0.0
5	0.28	15.3	5	0.00	0.0	5	0.00	0.0
6	1.74	18.0	6	0.99	17.1	6	0.10	3.4
7	15.50	24.5	7	4.80	21.4	7	1.30	3.6
8	200.28	81.0	8	37.89	28.7	8	26.59	7.1
9	2718.45	557.3	9	697.24	45.8	9	1086.21	34.0
	cvc4-real-clocks-int-val-$\mathcal{E}^2$			*z3-real-clocks-int-val-$\mathcal{E}^2$*			*yices-real-clocks-int-val-$\mathcal{E}^2$*	
3	0.00	0.0	3	0.00	0.0	3	0.00	0.0
4	0.00	0.0	4	0.00	0.0	4	0.00	0.0
5	0.30	14.4	5	0.00	0.0	5	0.00	0.0
6	2.19	16.9	6	1.39	18.3	6	0.10	3.0
7	18.79	22.4	7	5.50	21.2	7	1.90	4.0
8	199.17	68.0	8	50.99	28.5	8	38.49	8.3
9	2906.83	579.2	9	694.27	42.4	9	1416.26	40.9
	cvc4-real-clocks-bv-val-$\mathcal{E}^3$			*z3-real-clocks-bv-val-$\mathcal{E}^3$*			*yices-bv-clocks-bv-val-$\mathcal{E}^3$*	
3	0.00	0.0	3	0.00	0.0	3	0.00	0.0
4	0.00	0.0	4	0.09	15.7	4	0.00	0.0
5	0.39	17.4	5	0.49	16.4	5	0.10	4.0
6	3.39	21.3	6	7.19	19.6	6	3.89	5.5
7	36.00	32.1	7	112.19	27.9	7	74.48	12.2
8	512.64	147.1	8	1415.20	49.8	8	TIMEOUT	70.9
9	TIMEOUT	597.3	9	TIMEOUT	68.5	9	TIMEOUT	96.4
	cvc4-real-clocks-bv-val-$\mathcal{E}^2$			*z3-real-clocks-bv-val-$\mathcal{E}^2$*			*yices-bv-clocks-bv-val-$\mathcal{E}^2$*	
3	0.00	0.0	3	0.00	0.0	3	0.00	0.0
4	0.09	15.5	4	0.10	15.7	4	0.00	0.0
5	0.59	16.6	5	0.79	16.9	5	0.10	4.0
6	5.20	19.8	6	8.29	20.3	6	3.90	5.6
7	56.09	32.2	7	97.79	25.9	7	76.59	12.0
8	1277.46	274.5	8	2441.23	56.9	8	TIMEOUT	74.3
9	TIMEOUT	554.1	9	TIMEOUT	62.6	9	TIMEOUT	96.0
	cvc4-bv-clocks-int-val-$\mathcal{E}^3$			*z3-bv-clocks-int-val-$\mathcal{E}^3$*			*boolector-bv-clocks-bv-val-$\mathcal{E}^3$*	
3	0.00	0.0	3	0.00	0.0	3	0.00	0.0
4	0.00	0.0	4	0.00	0.0	4	0.10	4.3
5	0.20	15.0	5	0.18	16.9	5	0.69	5.6
6	1.40	17.8	6	1.69	18.6	6	5.39	10.0
7	13.09	26.5	7	13.98	22.9	7	94.29	34.2
8	141.88	80.7	8	270.86	31.7	8	1491.46	95.2
9	1811.85	642.8	9	1755.95	57.1	9	TIMEOUT	140.7
	cvc4-bv-clocks-int-val-$\mathcal{E}^2$			*z3-bv-clocks-int-val-$\mathcal{E}^2$*			*boolector-bv-clocks-bv-val-$\mathcal{E}^2$*	
3	0.00	0.0	3	0.00	0.0	3	0.00	0.0
4	0.00	0.0	4	0.00	0.0	4	0.09	4.6
5	0.19	14.5	5	0.29	16.5	5	0.69	6.1
6	1.99	17.6	6	2.69	18.0	6	4.30	9.6
7	23.89	39.6	7	26.67	21.2	7	86.49	29.1
8	600.84	359.4	8	394.60	32.4	8	1122.07	87.4
9	TIMEOUT	978.5	9	2862.76	54.1	9	TIMEOUT	133.8
	cvc4-bv-clocks-bv-val-$\mathcal{E}^3$			*z3-bv-clocks-bv-val-$\mathcal{E}^3$*				
3	0.00	0.0	3	0.00	0.0			
4	0.00	0.0	4	0.00	0.0			
5	0.20	22.4	5	0.10	14.8			
6	2.69	40.9	6	3.10	25.1			
7	116.99	280.2	7	70.59	40.8			
8	TIMEOUT	1911.2	8	3521.01	145.8			
9	TIMEOUT	2022.0	9	TIMEOUT	133.6			
	cvc4-bv-clocks-bv-val-$\mathcal{E}^2$			*z3-bv-clocks-bv-val-$\mathcal{E}^2$*				
3	0.03	15.2	3	0.10	12.6			
4	0.00	0.0	4	0.10	13.1			
5	0.19	20.1	5	0.79	14.7			
6	2.98	40.5	6	10.59	26.1			
7	188.26	307.8	7	252.13	40.5			
8	TIMEOUT	1801.0	8	TIMEOUT	101.8			
9	TIMEOUT	1785.3	9	TIMEOUT	114.4			

Table G.2: Elapsed time and memory usage for the concurrency challenge benchmark.

	$\mathcal{E}^3$				$\mathcal{E}^2$			
N	Conflicts		Lemmas	Propagations	Conflicts		Lemmas	Propagations
	SAT	Arith			SAT	Arith		
	cvc4-real-clocks-int-val							
3	16	9	72	43	21	14	72	81
4	74	49	112	273	96	63	112	492
5	753	303	162	1425	689	312	162	2566
6	4186	1480	222	7075	4583	1564	222	11888
7	29117	9860	292	36588	29691	9741	292	62555
8	251439	72955	372	107199	229774	72134	372	414910
9	2167878	673152	462	1783563	2143767	679653	462	3827080
	z3-real-clocks-int-val							
3	32	10	N/A	1384	37	10	N/A	1390
4	139	35	N/A	8138	114	41	N/A	7791
5	668	220	N/A	60436	753	201	N/A	66566
6	9270	2013	N/A	1381470	11848	2837	N/A	1690658
7	48012	14714	N/A	6374495	41440	12303	N/A	6184846
8	247414	80227	N/A	36559548	276715	88370	N/A	45395696
9	2622840	893610	N/A	445972388	2484169	850540	N/A	466603553
	yices2-real-clocks-int-val							
3	31	4	N/A	1343	34	8	N/A	1579
4	97	31	N/A	6285	111	49	N/A	6537
5	701	209	N/A	52717	714	220	N/A	51934
6	4262	1437	N/A	358039	4389	1379	N/A	399016
7	34452	10243	N/A	3170525	40406	10789	N/A	4091721
8	287873	79424	N/A	33941794	344728	96090	N/A	41105429
9	2971496	782923	N/A	403398242	3482115	877904	N/A	480198572

Table G.3: Concurrency challenge benchmark results for pure arithmetic (`QF_LIRA`).

G.2.1 Experimental results using interval theory lemmas

Tables G.5 and G.6 give the detailed experimental results summarized in Section 5.7.

$\mathcal{E}^3$							$\mathcal{E}^2$						
Time (*min*)	Conflicts SAT	BV	Arith	Lemmas Arith	Propagations BV	Arith	Time (*min*)	Conflicts SAT	BV	Arith	Lemmas Arith	Propagations BV	Arith
cvc4-bv-clocks-int-val ($N = 9$)													
8.4	2764998	0	11447	258	11519994	534213	217.5	8273911	3301	12457	258	42797338	1123180
z3-bv-clocks-int-val ($N = 9$)													
3.6	1723269	14	447	N/A	297426478		11.3	3144329	6	546	N/A	514531142	
mathsat5-bv-clocks-int-val ($N = 9$)													
1.7	734533	0	1115	N/A	108576850		1.7	757934	0	1114	N/A	149225806	

Table G.5: Concurrency challenge benchmark results for mixed BV/arithmetic with interval theory lemmas.

	$\mathcal{E}^3$				$\mathcal{E}^2$				
Time (*min*)	Conflicts SAT	Arith	Lemmas	Propagations	Time (*min*)	Conflicts SAT	Arith	Lemmas	Propagations
	cvc4-real-clocks-int-val ($N = 9$)								
1.5	413846	1981	560	112124	3.5	725911	2219	560	800500
	z3-real-clocks-int-val ($N = 9$)								
5.5	2268294	1200	N/A	437690276	6.4	2273053	1272	N/A	466030695
	yices2-real-clocks-int-val ($N = 9$)								
4.4	1384029	39614	N/A	200144167	2.5	981049	40712	N/A	142968657
	mathsat5-real-clocks-int-val ($N = 9$)								
1.6	707694	1182	N/A	138631783	2.9	1086485	1259	N/A	230430545

Table G.6: Concurrency challenge benchmark results for pure arithmetic (`QF_LIRA`) with interval theory lemma.

Bibliography

[AAA+15] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas, *Stateless model checking for TSO and PSO*, Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015) (Christel Baier and Cesare Tinelli, eds.), vol. 9035, Springer, 2015, pp. 353–367, doi:10.1007/978-3-662-46681-0_28.

[AAB05] Amitanand Aiyer, Lorenzo Alvisi, and Rida A. Bazzi, *On the availability of non-strict quorum systems*, Proceedings of the 19th International Conference on Distributed Computing (DISC 2005) (Pierre Fraigniaud, ed.), Lecture Notes in Computer Science, vol. 3724, Springer, 2005, pp. 48–62, doi:10.1007/11561927_6.

[AAJS14] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas, *Optimal dynamic partial order reduction*, Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014), ACM, 2014, pp. 373–384, doi:10.1145/2535838.2535845.

[AB10] Sarita V. Adve and Hans-J. Boehm, *Memory models: A case for rethinking parallel languages and hardware*, Communications of the ACM **53** (2010), no. 8, 90–101, doi:10.1145/1787234.1787255.

[ABH+01] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani, *Partial-order reduction in symbolic state-space exploration*, Formal Methods in System Design **18** (2001), no. 2, 97–116, doi:10.1023/A:1008767206905.

[Adv10] Sarita Adve, *Data races are evil with no exceptions: Technical perspective*, Communications of the ACM **53** (2010), no. 11, 84–84, doi:10.1145/1839676.1839697.

[AHMN91] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer, *Detecting data races on weak memory systems*, Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA 1991), ACM, 1991, pp. 234–243, doi:10.1145/115952.115976.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig, *Partial orders for efficient bounded model checking of concurrent software*, Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013) (Natasha Sharygina and Helmut Veith, eds.), Lecture Notes in Computer Science, vol. 8044, Springer, 2013, pp. 141–157, doi:10.1007/978-3-642-39799-8_9.

[ALS+10] Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie, *What consistency does your key-value store actually provide?*, Proceedings of the 6th International Conference on Hot Topics in System Dependability (HotDep 2010), USENIX Association, 2010, pp. 1–16.

[AM13] Aws Albarghouthi and Kenneth L. McMillan, *Beautiful interpolants*, Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013) (Natasha Sharygina and Helmut Veith, eds.), Lecture Notes in Computer Science, vol. 8044, Springer, 2013, pp. 313–329, doi:10.1007/978-3-642-39799-8_22.

[AMP00] Rajeev Alur, Ken McMillan, and Doron Peled, *Model-checking of correctness conditions for concurrent objects*, Information and Computation **160** (2000), no. 1-2, 167–188, doi:10.1006/inco.1999.2847.

[AMSS11] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell, *Litmus: Running tests against hardware*, Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011) (Parosh Aziz Abdulla and K. Rustan M. Leino, eds.), Lecture Notes in Computer Science, vol. 6605, Springer, 2011, pp. 41–44, doi:10.1007/978-3-642-19835-9_5.

[AMSS12] ———, *Fences in weak memory models (extended version)*, Formal Methods in System Design **40** (2012), no. 2, 170–205, doi:10.1007/s10703-011-0135-z.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig, *Herding cats: Modelling, simulation, testing, and data mining for weak memory*, ACM Transactions on Programming Languages and Systems (TOPLAS) **36** (2014), no. 2, 7:1–7:74, doi:10.1145/2627752.

[Ana14] Saswat Anand, *A bibliography of papers on symbolic execution technique and its applications*, https://sites.google.com/site/symexbib/, November 2014.

[ARR+07] Daphna Amit, Noam Rinetzky, Thomas Reps, Mooly Sagiv, and Eran Yahav, *Comparison under abstraction for verifying linearizability*, Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007) (Werner Damm and Holger Hermanns, eds.), Lecture Notes in Computer Science, vol. 4590, Springer, 2007, pp. 477–490, doi:10.1007/978-3-540-73368-3_49.

[AS09] Gilles Audemard and Laurent Simon, *Predicting learnt clauses quality in modern SAT solvers*, Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009), Morgan Kaufmann Publishers, 2009, pp. 399–404.

[BA12] Hans-J. Boehm and Sarita V. Adve, *You don't know jack about shared variables or memory models*, Communications of the ACM **55** (2012), no. 2, 48–54, doi:10.1145/2076450.2076465.

[BAM07] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin, *CheckFence: Checking consistency of concurrent data types on relaxed memory models*, Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007), ACM, 2007, pp. 12–21, doi:10.1145/1250734.1250737.

[BB09] Robert Brummayer and Armin Biere, *Boolector: An efficient SMT solver for bit-vectors and arrays*, Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009) (Stefan Kowalewski and Anna Philippou,

eds.), Lecture Notes in Computer Science, vol. 5505, Springer, 2009, pp. 174–177, doi:10.1007/978-3-642-00768-2_16.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli, *CVC4*, Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011) (Ganesh Gopalakrishnan and Shaz Qadeer, eds.), Lecture Notes in Computer Science, vol. 6806, Springer, 2011, pp. 171–177, doi:10.1007/978-3-642-22110-1_14.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang, *Symbolic model checking:* 10^{20} *states and beyond*, Information and Computation **98** (1992), no. 2, 142–170, doi:10.1016/0890-5401(92)90017-A.

[BCM10] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley, *PACER: Proportional detection of data races*, Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010), ACM, 2010, pp. 255–268, doi:10.1145/1806596.1806626.

[BDdM08] Nikolaj Bjørner, Bruno Dutertre, and Leonardo de Moura, *Accelerating lemma learning using joins - DPPL(Join)*, Proceedings of the 15th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2008), 2008.

[BDG+14] Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening, *Deciding floating-point logic with abstract conflict driven clause learning*, Formal Methods in System Design **45** (2014), no. 2, 213–245, doi:10.1007/s10703-013-0203-7.

[BDM13] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer, *Checking and enforcing robustness against TSO*, Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP 2013) (Matthias Felleisen and Philippa Gardner, eds.), Lecture Notes in Computer Science, vol. 7792, Springer, 2013, pp. 533–553, doi:10.1007/978-3-642-37036-6_29.

[BDMT10] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan, *Line-up: A complete and automatic linearizability checker*, Proceedings

of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010), ACM, 2010, pp. 330–340, doi:10.1145/1806596.1806634.

[BEEH13] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza, *Verifying concurrent programs against sequential specifications*, Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP 2013) (Matthias Felleisen and Philippa Gardner, eds.), Lecture Notes in Computer Science, vol. 7792, Springer, 2013, pp. 290–309, doi:10.1007/978-3-642-37036-6_17.

[BEEH15] ———, *Tractable refinement checking for concurrent objects*, Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015), ACM, 2015, pp. 651–662, doi:10.1145/2676726.2677002.

[Bel05] Fabrice Bellard, *QEMU, a fast and portable dynamic translator*, Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC 2005), USENIX Association, 2005, pp. 41–41.

[Bey13] Dirk Beyer, *Second competition on software verification*, Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013) (Nir Piterman and Scott A. Smolka, eds.), Lecture Notes in Computer Science, vol. 7795, Springer, 2013, pp. 594–609, doi:10.1007/978-3-642-36742-7_43.

[BFT15] Clark Barrett, Pascal Fontaine, and Cesare Tinelli, *The SMT-LIB standard: Version 2.5*, Tech. report, Department of Computer Science, The University of Iowa, 2015, Available at http://smt-lib.org/.

[BGM13] Ella Bounimova, Patrice Godefroid, and David Molnar, *Billions and billions of constraints: Whitebox fuzz testing in production*, Proceedings of the 35th International Conference on Software Engineering (ICSE 2013), IEEE, May 2013, pp. 122–131, doi:10.1109/ICSE.2013.6606558.

[BGMY12] Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang, *Concurrent library correctness on the TSO memory model*, Proceedings of the 21st European Conference on Programming Languages and Systems (ESOP 2012) (Helmut Seidl, ed.), Lecture

Notes in Computer Science, vol. 7211, Springer, 2012, pp. 87–107, doi:10.1007/978-3-642-28869-2_5.

[BH08] Domagoj Babić and Alan J. Hu, *Exploiting shared structure in software verification conditions*, Proceedings of the Third International Haifa Verification Conference on Hardware and Software: Verification and Testing (HVC 2007) (Karen Yorav, ed.), Lecture Notes in Computer Science, vol. 4899, Springer, 2008, pp. 169–184, doi:10.1007/978-3-540-77966-7_15.

[BK92] Bard Bloom and Marta Z. Kwiatkowska, *Trade-offs in true concurrency: Pomsets and Mazurkiewicz traces*, Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics (MFPS 1991) (Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, eds.), Lecture Notes in Computer Science, vol. 598, Springer, 1992, pp. 350–375, doi:10.1007/3-540-55511-0_18.

[BK14] Peter Bailis and Kyle Kingsbury, *The network is reliable*, Queue **12** (2014), no. 7, 20:20–20:32, doi:10.1145/2639988.2639988.

[BLAM+08] Josh Berdine, Tal Lev-Ami, Roman Manevich, Ganesan Ramalingam, and Mooly Sagiv, *Thread quantification for concurrent shape analysis*, Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008) (Aarti Gupta and Sharad Malik, eds.), Lecture Notes in Computer Science, vol. 5123, Springer, 2008, pp. 399–413, doi:10.1007/978-3-540-70545-1_37.

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani, *A decade of software model checking with SLAM*, Communications of the ACM **54** (2011), no. 7, 68–76, doi:10.1145/1965724.1965743.

[BNOT06] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli, *Splitting on demand in SAT modulo theories*, Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2006) (Miki Hermann and Andrei Voronkov, eds.), Lecture Notes in Computer Science, vol. 4246, Springer, 2006, pp. 512–526, doi:10.1007/11916277_35.

[Bon11] Paolo Bonzini, *QEMU Developer Mailing List – qdev for programmers*, http://lists.nongnu.org/archive/html/qemu-devel/2011-07/msg00842.html, July 2011.

[BOS+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber, *Mathematizing C++ concurrency*, Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), ACM, 2011, pp. 55–66, doi:10.1145/1926385.1926394.

[BS97] Roberto J. Bayardo, Jr. and Robert C. Schrag, *Using CSP look-back techniques to solve real-world SAT instances*, Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI 1997), AAAI Press, 1997, pp. 203–208.

[BS08] Jacob Burnim and Koushik Sen, *Heuristics for scalable dynamic test generation*, Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), IEEE, September 2008, pp. 443–446, doi:10.1109/ASE.2008.69.

[BWB+11] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar, *Nitpicking C++ concurrency*, Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming (PPDP 2011), ACM, 2011, pp. 113–124, doi:10.1145/2003476.2003493.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler, *KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs*, Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008), USENIX Association, 2008, pp. 209–224.

[CDG05] Robert Colvin, Simon Doherty, and Lindsay Groves, *Verifying concurrent data structures by simulation*, Electronic Notes in Theoretical Computer Science **137** (2005), no. 2, 93–110, Proceedings of the REFINE 2005 Workshop, doi:10.1016/j.entcs.2005.04.026.

[CE05] Cristian Cadar and Dawson Engler, *Execution generated test cases: How to make systems code crash itself*, Proceedings of the 12th International

SPIN Symposium on Model Checking Software (SPIN 2005) (Patrice Godefroid, ed.), Lecture Notes in Computer Science, vol. 3639, Springer, 2005, pp. 2–23, doi:10.1007/11537328_2.

[CFF+06] David Currie, Xiushan Feng, Masahiro Fujita, Alan J. Hu, Mark Kwan, and Sreeranga Rajan, *Embedded software verification using symbolic execution and uninterpreted functions*, International Journal of Parallel Programming **34** (2006), no. 1, 61–91, doi:10.1007/s10766-005-0004-8.

[CGLM06] Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir, *Formal verification of a lazy concurrent list-based set algorithm*, Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006) (Thomas Ball and Robert B. Jones, eds.), Lecture Notes in Computer Science, vol. 4144, Springer, 2006, pp. 475–488, doi:10.1007/11817963_44.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea, *S2E: A platform for in-vivo multi-path analysis of software systems*, Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI), ACM, 2011, pp. 265–278, doi:10.1145/1950365.1950396.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda, *A tool for checking ANSI-C programs*, Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Kurt Jensen and Andreas Podelski, eds.), Lecture Notes in Computer Science, vol. 2988, Springer, 2004, pp. 168–176, doi:10.1007/978-3-540-24730-2_15.

[CKS05] Byron Cook, Daniel Kroening, and Natasha Sharygina, *Symbolic model checking for asynchronous boolean programs*, Proceedings of the 12th International SPIN Symposium on Model Checking Software (SPIN 2005) (Patrice Godefroid, ed.), Lecture Notes in Computer Science, vol. 3639, Springer, 2005, pp. 75–90, doi:10.1007/11537328_9.

[Cop96] James O. Coplien, C++ Gems (Stanley B. Lippman, ed.), SIGS Publications, Inc., New York, NY, USA, 1996, pp. 135–144.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, *Linux device drivers*, O'Reilly, 2005.

[dBW90] J. W. de Bakker and J. H. A. Warmerdam, *Metric pomset semantics for a concurrent language with recursion*, Proceedings of the LITP Spring School on Theoretical Computer Science on Semantics of Systems of Concurrent Processes (Irène Guessarian, ed.), Lecture Notes in Computer Science, vol. 469, 1990, pp. 21–49, doi:10.1007/3-540-53479-2_2.

[DHK13] Vijay D'Silva, Leopold Haller, and Daniel Kroening, *Abstract conflict driven learning*, Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), ACM, 2013, pp. 143–154, doi:10.1145/2429069.2429087.

[DLL62] Martin Davis, George Logemann, and Donald Loveland, *A machine program for theorem-proving*, Communications of the ACM **5** (1962), no. 7, 394–397, doi:10.1145/368273.368557.

[dMB08] Leonardo de Moura and Nikolaj Bjørner, *Z3: An efficient SMT solver*, Proceedings of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008) (C. R. Ramakrishnan and Jakob Rehof, eds.), Lecture Notes in Computer Science, vol. 4963, Springer, 2008, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[dMJ13] Leonardo de Moura and Dejan Jovanović, *A model-constructing satisfiability calculus*, Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2013) (Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, eds.), Lecture Notes in Computer Science, vol. 7737, 2013, pp. 1–12, doi:10.1007/978-3-642-35873-9_1.

[DP60] Martin Davis and Hilary Putnam, *A computing procedure for quantification theory*, Journal of the ACM **7** (1960), no. 3, 201–215, doi:10.1145/321033.321034.

[DP02] Brian A. Davey and Hilary A. Priestley, *Introduction to lattices and order*, second ed., Cambridge University Press, 2002, doi:10.1017/CBO9780511809088.

[DSW11] John Derrick, Gerhard Schellhorn, and Heike Wehrheim, *Mechanically verified proof obligations for linearizability*, ACM Transactions on Programming Languages and Systems (TOPLAS) **33** (2011), no. 1, 4:1–4:43, doi:10.1145/1889997.1890001.

[Dut14] Bruno Dutertre, *Yices 2.2*, Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2014) (Armin Biere and Roderick Bloem, eds.), Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 737–744, doi:10.1007/978-3-319-08867-9_49.

[DWS+12] Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer, *RADISH: Always-on sound and complete race detection in software and hardware*, Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA 2012), IEEE, June 2012, pp. 201–212, doi:10.1109/ISCA.2012.6237018.

[É02] Zoltán Ésik, *Axiomatizing the subsumption and subword preorders on finite and infinite partial words*, Theoretical Computer Science **273** (2002), no. 1-2, 225–248, doi:10.1016/S0304-3975(00)00442-4.

[EB05] Niklas Eén and Armin Biere, *Effective preprocessing in SAT through variable and clause elimination*, Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005) (Fahiem Bacchus and Toby Walsh, eds.), Lecture Notes in Computer Science, vol. 3569, Springer, 2005, pp. 61–75, doi:10.1007/11499107_5.

[Ers58] A. P. Ershov, *On programming of arithmetic operations*, Communications of the ACM **1** (1958), no. 8, 3–6, doi:10.1145/368892.368907.

[ETQ05] Tayfun Elmas, Serdar Tasiran, and Shaz Qadeer, *VYRD: Verifying concurrent programs by runtime refinement-violation detection*, Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005), ACM, 2005, pp. 27–37, doi:10.1145/1065010.1065015.

[FC06] Jean-Christophe Filliâtre and Sylvain Conchon, *Type-safe modular hash-consing*, Proceedings of the 2006 Workshop on ML (ML 2006), ACM, 2006, pp. 12–19, doi:10.1145/1159876.1159880.

[FF09] Cormac Flanagan and Stephen N. Freund, *FastTrack: Efficient and precise dynamic race detection*, Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), ACM, 2009, pp. 121–133, doi:10.1145/1542476.1542490.

[FG05] Cormac Flanagan and Patrice Godefroid, *Dynamic partial-order reduction for model checking software*, Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005), ACM, 2005, pp. 110–121, doi:10.1145/1040305.1040315.

[Fid88] Colin J. Fidge, *Timestamps in message-passing systems that preserve the partial ordering*, Proceedings of the 11th Australian Computer Science Conference (ACSC 1988), February 1988, pp. 56–66.

[Fis70] Peter C. Fishburn, *Intransitive indifference with unequal indifference intervals*, Journal of Mathematical Psychology **7** (1970), no. 1, 144–149, doi:10.1016/0022-2496(70)90062-3.

[FKL93] Joan Feigenbaum, Jeremy A. Kahn, and Carsten Lund, *Complexity results for POMSET languages*, SIAM Journal on Discrete Mathematics **6** (1993), no. 3, 432–442, doi:10.1137/0406035.

[FKP] *fkp2013 SV-COMP* `pthreads` *concurrency benchmark*, https://svn.sosy-lab.org/software/sv-benchmarks/trunk/c/pthread-lit/fkp2013_false-unreach-call.c?p=588.

[FKP13] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski, *Inductive data flow graphs*, Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), ACM, 2013, pp. 129–142, doi:10.1145/2429069.2429086.

[FLR11] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues, *Finding complex concurrency bugs in large multi-threaded applications*, Proceedings of the 6th Conference on Computer Systems (EuroSys 2011), ACM, 2011, pp. 215–228, doi:10.1145/1966445.1966465.

[GFYS07] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv, *Cartesian partial-order reduction*, Proceedings of the 14th International SPIN

Symposium on Model Checking Software (SPIN 2007), Springer, 2007, pp. 95–112, doi:10.1007/978-3-540-73370-6_8.

[GG91] Kourosh Gharachorloo and Phillip B. Gibbons, *Detecting violations of sequential consistency*, Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 1991), ACM, 1991, pp. 316–326, doi:10.1145/113379.113408.

[GG08] Malay K. Ganai and Aarti Gupta, *Efficient modeling of concurrent systems in BMC*, Proceedings of the 15th International SPIN Symposium on Model Checking Software (SPIN 2008) (Klaus Havelund, Rupak Majumdar, and Jens Palsberg, eds.), Lecture Notes in Computer Science, vol. 5156, Springer, 2008, pp. 114–133, doi:10.1007/978-3-540-85114-1_10.

[GHL13] Wojciech Golab, Jeremy Hurwitz, and Xiaozhou (Steve) Li, *On the k-atomicity-verification problem*, Proceedings of the IEEE 33rd International Conference on Distributed Computing Systems (ICDCS 2013), IEEE, 2013, pp. 591–600, doi:10.1109/ICDCS.2013.45.

[GHR+15] Ashutosh Gupta, Thomas A. Henzinger, Arjun Radhakrishna, Roopsha Samanta, and Thorsten Tarrach, *Succinct representation of concurrent trace sets*, Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015), ACM, 2015, pp. 433–444, doi:10.1145/2676726.2677008.

[Gis85] Jay L. Gischer, *Partial orders and the axiomatic theory of shuffle (pomsets)*, Ph.D. thesis, Stanford, CA, USA, 1985, AAI8506191.

[Gis88] ———, *The equational theory of pomsets*, Theoretical Computer Science **61** (1988), no. 2-3, 199–224, doi:10.1016/0304-3975(88)90124-7.

[GK97] Phillip B. Gibbons and Ephraim Korach, *Testing shared memories*, SIAM Journal on Computing **26** (1997), no. 4, 1208–1244, doi:10.1137/S0097539794279614.

[GKD06] Daniel Große, Ulrich Kühne, and Rolf Drechsler, *HW/SW co-verification of embedded systems using bounded model checking*, Pro-

ceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI 2006), ACM, 2006, pp. 43–48, doi:10.1145/1127908.1127920.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen, *DART: Directed automated random testing*, Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005), ACM, 2005, pp. 213–223, doi:10.1145/1065010.1065036.

[GKSS08] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman, *Satisfiability solvers*, Handbook of Knowledge Representation, Foundations of Artificial Intelligence, vol. 3, Elsevier, 2008, http://www.cs.rochester.edu/users/faculty/kautz/papers/SATSolvers-KR-Handbook.pdf, pp. 89–134.

[GL02] Seth Gilbert and Nancy Lynch, *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*, ACM SIGACT News **33** (2002), no. 2, 51–59, doi:10.1145/564585.564601.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy, *Memory consistency and event ordering in scalable shared-memory multiprocessors*, Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA 1990), ACM, 1990, pp. 15–26, doi:10.1145/325164.325102.

[GM15] Marco Gario and Andrea Micheli, *pySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms*, 13th International Workshop on Satisfiability Modulo Theories (SMT 2015), 2015.

[GNK+15] Ivan Gavran, Filip Niksic, Aditya Kanade, Rupak Majumdar, and Viktor Vafeiadis, *Rely/guarantee reasoning for asynchronous programs*, Proceedings of the 26th International Conference on Concurrency Theory (CONCUR 2015) (Luca Aceto and David de Frutos-Escrig, eds.), vol. 42, 2015, pp. 483–496, doi:10.4230/LIPIcs.CONCUR.2015.483.

[God91] Patrice Godefroid, *Using partial orders to improve automatic verification methods*, Proceedings of the 2nd International Workshop on Computer Aided Verification (CAV 1990) (Edmund M. Clarke and Robert P. Kurshan, eds.), Lecture Notes in Computer Science, vol. 531, Springer, 1991, pp. 176–185, doi:10.1007/BFb0023731.

[God96] ———, *Partial-order methods for the verification of concurrent systems: An approach to the state-explosion problem*, Springer, Secaucus, NJ, USA, 1996.

[Got74] Eiichi Goto, *Monocopy and associative algorithms in extended lisp*, Tech. Report TR-74-03, University of Toyko, 1974.

[Gra81] Jan Grabowski, *On partial languages*, Fundamenta Informaticae **4** (1981), no. 2, 427–498.

[GW93] Patrice Godefroid and Pierre Wolper, *Using partial orders for the efficient verification of deadlock freedom and safety properties*, Formal Methods in System Design **2** (1993), no. 2, 149–164, doi:10.1007/BF01383879.

[GW94] ———, *A partial approach to model checking*, Information and Computation **110** (1994), no. 2, 305–326, doi:10.1006/inco.1994.1035.

[GYS04] Ganesh Gopalakrishnan, Yue Yang, and Hemanthkumar Sivaraj, *QB or not QB: An efficient execution verification tool for memory orderings*, Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004) (Rajeev Alur and Doron A. Peled, eds.), Lecture Notes in Computer Science, vol. 3114, Springer, 2004, doi:10.1007/978-3-540-27813-9_31.

[HAMM14] Julien Henry, Mihail Asavoae, David Monniaux, and Claire Maïza, *How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics*, Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES 2014), ACM, 2014, pp. 43–52, doi:10.1145/2597809.2597817.

[HBJ+14] Liana Hadarean, Kshitij Bansal, Dejan Jovanović, Clark Barrett, and Cesare Tinelli, *A tale of two solvers: Eager and lazy approaches to bit-vectors*, Proceedings of the 16th International Conference on Computer Aided Verification (Armin Biere and Roderick Bloem, eds.), Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 680–695, doi:10.1007/978-3-319-08867-9_45.

[HC83] Timothy Hickey and Jacques Cohen, *Uniform random generation of strings in a context-free language*, SIAM Journal on Computing **12** (1983), no. 4, 645–655, doi:10.1137/0212044.

[HHK15] Liana Hadarean, Alex Horn, and Tim King, *A concurrency problem with exponential DPLL(T) proofs*, 13th International Workshop on Satisfiability Modulo Theories (SMT 2015), 2015.

[HHL+06] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit, *A lazy concurrent list-based set algorithm*, Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005) (James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, eds.), Lecture Notes in Computer Science, vol. 3974, Springer, 2006, pp. 3–16, doi:10.1007/11795490_3.

[HK15] Alex Horn and Daniel Kroening, *On partial order semantics for SAT/SMT-based symbolic encodings of weak memory concurrency*, Proceedings of the 35th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2015) (Susanne Graf and Mahesh Viswanathan, eds.), Lecture Notes in Computer Science, vol. 9039, Springer, 2015, pp. 19–34, doi:10.1007/978-3-319-19195-9_2.

[HMSW11] Tony Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman, *Concurrent Kleene algebra and its foundations*, The Journal of Logic and Algebraic Programming **80** (2011), no. 6, 266–296, doi:10.1016/j.jlap.2011.04.005.

[HN11] Pierre-Cyrille Héam and Cyril Nicaud, *Seed: An easy-to-use random generator of recursive data structures for testing*, Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST 2011), IEEE, March 2011, pp. 60–69, doi:10.1109/ICST.2011.31.

[Hoa69] C. A. R. Hoare, *An axiomatic basis for computer programming*, Communications of the ACM **12** (1969), no. 10, 576–580, doi:10.1145/363235.363259.

[HS12] Maurice Herlihy and Nir Shavit, *The art of multiprocessor programming, revised reprint*, first ed., Morgan Kaufmann Publishers, San Francisco, CA, USA, 2012.

[HTV+13] Alex Horn, Michael Tautschnig, Celina Val, Liaho Liang, Tom Melham, Jim Grundy, and Daniel Kroening, *Formal co-validation of low-level hardware/software interfaces*, Formal Methods in Computer-Aided Design (FMCAD 2013) (Barbara Jobstmann and Sandip Ray, eds.), IEEE, October 2013, pp. 121–128, doi:10.1109/FMCAD.2013.6679400.

[HvS12a] Tony Hoare and Stephan van Staden, *In praise of algebra*, Formal Aspects of Computing **24** (2012), no. 4-6, 423–431, doi:10.1007/s00165-012-0249-0.

[HvS12b] ———, *The laws of programming unify process calculi*, Proceedings of the 11th International Conference on Mathematics of Program Construction (MPC 2012) (Jeremy Gibbons and Pablo Nogueira, eds.), Lecture Notes in Computer Science, vol. 7342, Springer, 2012, pp. 7–22, doi:10.1007/978-3-642-31113-0_2.

[HvSM+14] Tony Hoare, Stephan van Staden, Bernhard Möller, Georg Struth, Jules Villard, Huibiao Zhu, and Peter O'Hearn, *Developments in concurrent Kleene algebra*, 14th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 2014), Lecture Notes in Computer Science, vol. 8428, Springer, Cham, 2014, pp. 1–18, doi:10.1007/978-3-319-06251-8_1.

[HW90] Maurice P. Herlihy and Jeannette M. Wing, *Linearizability: A correctness condition for concurrent objects*, ACM Transactions on Programming Languages and Systems (TOPLAS) **12** (1990), no. 3, 463–492, doi:10.1145/78969.78972.

[ISO12] ISO, *ISO/IEC 14882:2011 information technology — programming languages — C++*, International Organization for Standardization, February 2012.

[ISO14] ———, *ISO/IEC 14882:2014(E) information technology — programming languages — C++*, International Organization for Standardization, December 2014.

[ITF+14] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato, *Bounded model checking of multi-threaded C programs via lazy sequentialization*, Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2014) (Armin Biere and Roderick Bloem, eds.), Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 585–602, doi:10.1007/978-3-319-08867-9_39.

[Jip14] Peter Jipsen, *Concurrent Kleene algebra with tests*, Proceedings of the 14th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 2014) (Peter Höfner, Peter Jipsen, Wolfram Kahl, and Martin Eric Müller, eds.), Lecture Notes in Computer Science, vol. 8428, Springer, 2014, pp. 37–48, doi:10.1007/978-3-319-06251-8_3.

[JR16] Alan Jeffrey and James Riely, *On thin air reads: Towards an event structures model of relaxed memory*, Proceedings of IEEE Logic in Computer Science, LICS, 2016.

[Kah87] Gilles Kahn, *Natural semantics*, 4th Annual Symposium on Theoretical Aspects of Computer Sciences (STACS 1987) (Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, eds.), Lecture Notes in Computer Science, vol. 247, Springer, 1987, pp. 22–39, doi:10.1007/BFb0039592.

[KCC10] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea, *Testing closed-source binary device drivers with DDT*, Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC 2010), USENIX Association, 2010, pp. 12–12.

[KCY03] Daniel Kroening, Edmund Clarke, and Karen Yorav, *Behavioral consistency of C and Verilog programs using bounded model checking*, Proceedings of the 40th Annual Design Automation Conference (DAC 2003), ACM, 2003, pp. 368–371, doi:10.1145/775832.775928.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood, *seL4: Formal verification of an OS kernel*, Proceedings of

the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP 2009), ACM, 2009, pp. 207–220, doi:10.1145/1629575.1629596.

[KGS06] Vineet Kahlon, Aarti Gupta, and Nishant Sinha, *Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions*, Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006) (Thomas Ball and Robert B. Jones, eds.), Lecture Notes in Computer Science, vol. 4144, Springer, 2006, pp. 286–299, doi:10.1007/11817963_28.

[Kin76] James C. King, *Symbolic execution and program testing*, Communications of the ACM **19** (1976), no. 7, 385–394, doi:10.1145/360248.360252.

[Kin14] Kyle Kingsbury, *Computational techniques in Knossos*, https://aphyr.com/posts/314-computational-techniques-in-knossos, May 2014.

[KLM+02] Robert P. Kurshan, Vladimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün, *Combining software and hardware verification techniques*, Formal Methods in System Design **21** (2002), no. 3, 251–280, doi:10.1023/A:1020383505582.

[KPV03] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser, *Generalized symbolic execution for model checking and testing*, Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003) (Hubert Garavel and John Hatcliff, eds.), Lecture Notes in Computer Science, vol. 2619, Springer, 2003, pp. 553–568, doi:10.1007/3-540-36577-X_40.

[KRB+15] Avi Kivity, Ademar de Souza Jr Reis, Paolo Bonzini, Marcel Apfelbaum, Peter Maydell, and Fam Zheng, *The QEMU memory API*, http://git.qemu.org/?p=qemu.git;a=blob;f=docs/memory.txt, February 2015.

[KS08] Daniel Kroening and Ofer Strichman, *Decision procedures: An algorithmic point of view*, first ed., Springer, 2008.

[KT14] Daniel Kroening and Michael Tautschnig, *CBMC - C bounded model checker - (competition contribution)*, Proceedings of the 20th Interna-

tional Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014) (Erika Ábrahám and Klaus Havelund, eds.), Lecture Notes in Computer Science, vol. 8413, 2014, pp. 389–391, doi:10.1007/978-3-642-54862-8_26.

[KWG09] Vineet Kahlon, Chao Wang, and Aarti Gupta, *Monotonic partial order reduction: An optimal symbolic partial order reduction technique*, Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009) (Ahmed Bouajjani and Oded Maler, eds.), Lecture Notes in Computer Science, vol. 5643, Springer, 2009, pp. 398–413, doi:10.1007/978-3-642-02658-4_31.

[KYKS12] Kyung Hee Kim, Tuba Yavuz-Kahveci, and Beverly A. Sanders, *JRF-E: using model checking to give advice on eliminating memory model-related bugs*, Automated Software Engineering **19** (2012), no. 4, 491–530, doi:10.1007/s10515-012-0109-4.

[Lam78] Leslie Lamport, *Time, clocks, and the ordering of events in a distributed system*, Communications of the ACM **21** (1978), no. 7, 558–565, doi:10.1145/359545.359563.

[Lam79a] ———, *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Transactions on Computers **C-28** (1979), no. 9, 690–691, doi:10.1109/TC.1979.1675439.

[Lam79b] ———, *A new approach to proving the correctness of multiprocess programs*, ACM Transactions on Programming Languages and Systems (TOPLAS) **1** (1979), no. 1, 84–97, doi:10.1145/357062.357068.

[Lam80] ———, *Corrigendum: A new approach to proving the correctness of multiprocess programs*, ACM Transactions on Programming Languages and Systems (TOPLAS) **2** (1980), no. 1, 134, doi:10.1145/357084.357093.

[Lam97] ———, *How to make a correct multiprocess program execute correctly on a multiprocessor*, IEEE Transactions on Computers **46** (1997), no. 7, 779–782, doi:10.1109/12.599898.

[LCL+13] Yang Liu, Wei Chen, Yanhong A. Liu, Jun Sun, Shao Jie Zhang, and Jin Song Dong, *Verifying linearizability via optimized refinement check-*

ing, IEEE Transactions on Software Engineering **39** (2013), no. 7, 1018–1039, doi:10.1109/TSE.2012.82.

[Lig11] Anthony Liguori, *QEMU Wiki – QOM*, http://wiki.qemu.org/Features/QOM, September 2011.

[LMPS05] Mirko Loghi, Tiziana Margaria, Graziano Pravadelli, and Bernhard Steffen, *Dynamic and formal verification of embedded systems: A comparative survey*, International Journal of Parallel Programming **33** (2005), no. 6, 585–611, doi:10.1007/s10766-005-8911-2.

[Low15] Gavin Lowe, *Testing for linearizability*, Under submission. http://www.cs.ox.ac.uk/people/gavin.lowe/LinearizabiltyTesting/, 2015.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou, *Learning from mistakes: A comprehensive study on real world concurrency bug characteristics*, Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII), ACM, 2008, pp. 329–339, doi:10.1145/1346281.1346323.

[LQR09] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić, *Static and precise detection of concurrency errors in systems code using SMT solvers*, Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009) (Ahmed Bouajjani and Oded Maler, eds.), Lecture Notes in Computer Science, vol. 5643, Springer, 2009, pp. 509–524, doi:10.1007/978-3-642-02658-4_38.

[LS14] Michael R. Laurence and Georg Struth, *Completeness theorems for Bi-Kleene algebras and series-parallel rational pomset languages*, Proceedings of the 14th International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 2014) (Peter Höfner, Peter Jipsen, Wolfram Kahl, and Martin Eric Müller, eds.), Lecture Notes in Computer Science, vol. 8428, Springer, 2014, pp. 65–82, doi:10.1007/978-3-319-06251-8_5.

[LST03] Flavio Lerda, Nishant Sinha, and Michael Theobald, *Symbolic model checking of software*, Electronic Notes in Theoretical Computer Science **89** (2003), no. 3, 480–498, Workshop on Software Model Checking (SoftMC 2003), doi:10.1016/S1571-0661(05)80008-8.

[LWF+15] Anthony Liguori, Kevin Wolf, Andreas Färber, Markus Armbruster, Alex Bligh, Stefan Hajnoczi, Peter Maydell, John Snow, and Marc Marí, *The QTest API*, http://git.qemu-project.org/?p=qemu.git;a=blob;f=tests/libqtest.h, May 2015.

[LXB+10] Juncao Li, Fei Xie, Thomas Ball, Vladimir Levin, and Con McGarvey, *An automata-theoretic approach to hardware/software co-verification*, Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE 2010) (David S. Rosenblum and Gabriele Taentzer, eds.), Lecture Notes in Computer Science, vol. 6013, Springer, 2010, pp. 248–262, doi:10.1007/978-3-642-12029-9_18.

[Mat89] Friedemann Mattern, *Virtual time and global states of distributed systems*, Proceedings Workshop on Parallel and Distributed Algorithms, 1989, pp. 215–226.

[Maz87] Antoni Mazurkiewicz, *Trace theory*, pp. 278–324, Springer, 1987, doi:10.1007/3-540-17906-2_30.

[McM93] Kenneth L. McMillan, *Symbolic model checking*, Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[Mey12] Scott Meyers, *Universal references in C++11*, https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers, November 2012.

[MHMS+12] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams, *An axiomatic memory model for POWER multiprocessors*, Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012) (P. Madhusudan and Sanjit A. Seshia, eds.), Lecture Notes in Computer Science, vol. 7358, Springer, 2012, pp. 495–512, doi:10.1007/978-3-642-31424-7_36.

[Mil80] Robin Milner, *A calculus of communicating systems*, Lecture Notes in Computer Science, vol. 92, Springer, 1980, doi:10.1007/3-540-10235-3.

[MKS09] Kenneth L. McMillan, Andreas Kuehlmann, and Mooly Sagiv, *Generalizing DPLL to richer logics*, Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009) (Ahmed Bouajjani and Oded Maler, eds.), Lecture Notes in Computer Science, vol. 5643, Springer, 2009, pp. 462–476, doi:10.1007/978-3-642-02658-4_35.

[Moh11] Igor Mohor, *Opencores ethernet MAC 10/100 Mbps*, http://opencores.org/project,ethmac, Jul. 2011.

[Mon07] David Monniaux, *Verification of device drivers and intelligent controllers: A case study*, Proceedings of the 7th IEEE/ACM International Conference on Embedded Software (EMSOFT 2007), ACM, 2007, pp. 30–36, doi:10.1145/1289927.1289937.

[MSS96] João P. Marques-Silva and Karem A. Sakallah, *Grasp—a new search algorithm for satisfiability*, Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 1996), IEEE, 1996, pp. 220–227, doi:10.1109/ICCAD.1996.569607.

[MSS99] ———, *Grasp: a search algorithm for propositional satisfiability*, IEEE Transactions on Computers **48** (1999), no. 5, 506–521, doi:10.1109/12.769433.

[ND13] Brian Norris and Brian Demsky, *CDSchecker: Checking concurrent data structures written with C/C++ atomics*, Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2013), ACM, 2013, pp. 131–150, doi:10.1145/2509136.2509514.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli, *Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)*, Journal of the ACM **53** (2006), no. 6, 937–977, doi:10.1145/1217856.1217859.

[NPW81] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel, *Petri nets, event structures and domains, part I*, Theoretical Computer Science **13** (1981), no. 1, 85–108, Special Issue Semantics of Concurrent Computation, doi:10.1016/0304-3975(81)90112-2.

[NWSK11] Minh D. Nguyen, Markus Wedler, Dominik Stoffel, and Wolfgang Kunz, *Formal hardware/software co-verification by interval property checking with abstraction*, Proceedings of the 48th Design Automation Conference (DAC 2011), ACM, 2011, pp. 510–515, doi:10.1145/2024724.2024843.

[OG76] Susan Owicki and David Gries, *Verifying properties of parallel programs: An axiomatic approach*, Communications of the ACM **19** (1976), no. 5, 279–285, doi:10.1145/360051.360224.

[Oka98] Chris Okasaki, *Purely functional data structures*, Cambridge University Press, 1998.

[OPVH15] Peter W. O'Hearn, Rasmus L. Petersen, Jules Villard, and Akbar Hussain, *On the relation between concurrent separation logic and concurrent Kleene algebra*, Journal of Logical and Algebraic Methods in Programming **84** (2015), no. 3, 285–302, doi:10.1016/j.jlamp.2014.08.002.

[Pel93] Doron Peled, *All from one, one for all: on model checking using representatives*, Proceedings of the 5th International Conference on Computer Aided Verification (CAV 1993) (Costas Courcoubetis, ed.), Lecture Notes in Computer Science, vol. 697, Springer, 1993, pp. 409–423, doi:10.1007/3-540-56922-7_34.

[Pel96] ———, *Combining partial order reductions with on-the-fly model-checking*, Formal Methods in System Design **8** (1996), no. 1, 39–64, doi:10.1007/BF00121262.

[Pet66] Carl Adam Petri, *Communication with automata*, Ph.D. thesis, Universität Hamburg, 1966.

[PG12] Michael Pradel and Thomas R. Gross, *Fully automatic and precise detection of thread safety violations*, Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012), ACM, 2012, pp. 521–530, doi:10.1145/2254064.2254126.

[PG13] ———, *Automatic testing of sequential and concurrent substitutability*, Proceedings of the 2013 International Conference on Software Engineer-

ing (ICSE 2013), IEEE, May 2013, pp. 282–291, doi:10.1109/ICSE.2013.6606574.

[PICW04] Ganapathy Parthasarathy, Madhu K. Iyer, Kwang-Ting Cheng, and Li-C. Wang, *An efficient finite-domain constraint solver for circuits*, Proceedings of the 41st Design Automation Conference (DAC 2004), ACM, July 2004, pp. 212–217, doi:10.1145/996566.996628.

[PPH+11] Gilles Pokam, Cristiano Pereira, Shiliang Hu, Ali-Reza Adl-Tabatabai, Justin Gottschlich, Jungwoo Ha, and Youfeng Wu, *CoreRacer: A practical memory race recorder for multicore x86 TSO processors*, Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44), ACM, 2011, pp. 216–225, doi:10.1145/2155620.2155646.

[PPS16] Jean Pichon-Pharabod and Peter Sewell, *A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions*, Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016), ACM, 2016, pp. 622–633, doi:10.1145/2837614.2837616.

[Pra86] Vaughan Pratt, *Modeling concurrency with partial orders*, International Journal of Parallel Programming **15** (1986), no. 1, 33–71, doi:10.1007/BF01379149.

[Rab78] Issie Rabinovitch, *The dimension of semiorders*, Journal of Combinatorial Theory, Series A **25** (1978), no. 1, 50–61, doi:10.1016/0097-3165(78)90030-4.

[Res99] Pedro Resende, *Quantales, concurrent observations and event structures*, Preprint, Departamento de Matemática, Instituto Superior Técnico, Lisboa, 1999.

[RG05] Ishai Rabinovitz and Orna Grumberg, *Bounded model checking of concurrent programs*, Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005) (Kousha Etessami and Sriram K. Rajamani, eds.), Lecture Notes in Computer Science, vol. 3576, Springer, 2005, pp. 82–97, doi:10.1007/11513988_9.

[RHB97] A. W. Roscoe, C. A. R. Hoare, and Richard Bird, *The theory and practice of concurrency*, Prentice Hall International Series in Computer Science, Upper Saddle River, NJ, USA, 1997.

[RSSK15] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening, *Unfolding-based partial order reduction*, Proceedings of the 26th International Conference on Concurrency Theory (CONCUR 2015) (Luca Aceto and David de Frutos Escrig, eds.), vol. 42, 2015, pp. 456–469, doi:10.4230/LIPIcs.CONCUR.2015.456.

[SBA+11] Ohad Shacham, Nathan Bronson, Alex Aiken, Mooly Sagiv, Martin Vechev, and Eran Yahav, *Testing atomicity of composed concurrent operations*, Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2011), ACM, 2011, pp. 51–64, doi:10.1145/2048066.2048073.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov, *ThreadSanitizer: Data race detection in practice*, Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA 2009), ACM, 2009, pp. 62–71, doi:10.1145/1791194.1791203.

[SOK01] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov, *Beyond softnet*, Proceedings of the 5th Annual Linux Showcase & Conference (ALS 2001), vol. 5, USENIX Association, 2001, pp. 18–18.

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams, *Understanding POWER multiprocessors*, Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011), ACM, 2011, pp. 175–186, doi:10.1145/1993498.1993520.

[SSB02] Ofer Strichman, Sanjit A. Seshia, and Randal E. Bryant, *Deciding separation formulas with SAT*, Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002) (Ed Brinksma and KimGuldstrand Larsen, eds.), Lecture Notes in Computer Science, vol. 2404, Springer, 2002, pp. 209–222, doi:10.1007/3-540-45657-0_16.

[SSO+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen, *x86-TSO: A rigorous and usable programmer's*

model for x86 multiprocessors, Communications of the ACM **53** (2010), no. 7, 89–97, doi:10.1145/1785414.1785443.

[SW10] Nishant Sinha and Chao Wang, *Staged concurrent program analysis*, Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2010), ACM, 2010, pp. 47–56, doi:10.1145/1882291.1882301.

[TBL12] Tim Todman, Peter Boehm, and Wayne Luk, *Verification of streaming hardware and software codesigns*, Proceedings of the 2012 International Conference on Field-Programmable Technology (FPT 2012), IEEE, December 2012, pp. 147–150, doi:10.1109/FPT.2012.6412127.

[Tei12] Jürgen Teich, *Hardware/software codesign: The past, the present, and predicting the future*, Proceedings of the IEEE **100** (2012), no. Special Centennial Issue, 1411–1430, doi:10.1109/JPROC.2011.2182009.

[TIF+15] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato, *Verifying concurrent programs by memory unwinding*, Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015) (Christel Baier and Cesare Tinelli, eds.), vol. 9035, Springer, 2015, pp. 551–565, doi:10.1007/978-3-662-46681-0_52.

[TR12] Aditya Thakur and Thomas Reps, *A method for symbolic computation of abstract operations*, Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012) (P. Madhusudan and Sanjit A. Seshia, eds.), Lecture Notes in Computer Science, vol. 7358, Springer, 2012, pp. 174–192, doi:10.1007/978-3-642-31424-7_17.

[TVD10] Emina Torlak, Mandana Vaziri, and Julian Dolby, *MemSAT: Checking axiomatic specifications of memory models*, Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010), ACM, 2010, pp. 341–350, doi:10.1145/1806596.1806635.

[Vaf09] Viktor Vafeiadis, *Shape-value abstraction for verifying linearizability*, Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009) (Neil D. Jones

and Markus Müller-Olm, eds.), Lecture Notes in Computer Science, vol. 5403, Springer, 2009, pp. 335–348, doi:10.1007/978-3-540-93900-9_27.

[Vaf10] ———, *Automatically proving linearizability*, Proceedings of the 22nd International Conference on Computer Aided Verification (CAV 2010) (Tayssir Touili, Byron Cook, and Paul Jackson, eds.), Lecture Notes in Computer Science, vol. 6174, Springer, 2010, pp. 450–464, doi:10.1007/978-3-642-14295-6_40.

[Val92] Antti Valmari, *A stubborn attack on state explosion*, Formal Methods in System Design **1** (1992), no. 4, 297–322, doi:10.1007/BF00709154.

[Vel96] Todd Veldhuizen, C++ Gems (Stanley B. Lippman, ed.), SIGS Publications, Inc., New York, NY, USA, 1996, pp. 459–473.

[vGP09] Rob J. van Glabbeek and Gordon D. Plotkin, *Configuration structures, event structures and Petri nets*, Theoretical Computer Science **410** (2009), no. 41, 4111–4159, Festschrift for Mogens Nielsen's 60th birthday, doi:10.1016/j.tcs.2009.06.014.

[VPK04] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid, *Test input generation with Java PathFinder*, Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004), ACM, 2004, pp. 97–107, doi:10.1145/1007512.1007526.

[vRZ+10] Pavol Černý, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur, *Model checking of linearizability of concurrent list implementations*, Proceedings of the 22nd International Conference on Computer Aided Verification (CAV 2010) (Tayssir Touili, Byron Cook, and Paul Jackson, eds.), Lecture Notes in Computer Science, vol. 6174, Springer, 2010, pp. 465–479, doi:10.1007/978-3-642-14295-6_41.

[vVZN+11] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell, *Relaxed-memory concurrency and verified compilation*, Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011), ACM, 2011, pp. 43–54, doi:10.1145/1926385.1926393.

[VYY09] Martin Vechev, Eran Yahav, and Greta Yorsh, *Experience with model checking linearizability*, Proceedings of the 16th International SPIN Symposium on Model Checking Software (SPIN 2009) (Corina S. Păsăreanu, ed.), Lecture Notes in Computer Science, vol. 5578, Springer, 2009, pp. 261–278, doi:10.1007/978-3-642-02652-2_21.

[WG93] Jeannette M. Wing and Chun Gong, *Testing and verifying concurrent objects*, Journal of Parallel and Distributed Computing **17** (1993), no. 1-2, 164–182, doi:10.1006/jpdc.1993.1015.

[Win82] Glynn Winskel, *Event structure semantics for CCS and related languages*, Proceedings of the 9th International Colloquium on Automata, Languages and Programming (ICALP 1982) (Mogens Nielsen and Erik Meineche Schmidt, eds.), Lecture Notes in Computer Science, vol. 140, Springer, 1982, pp. 561–576, doi:10.1007/BFb0012800.

[WKGG09] Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta, *Symbolic predictive analysis for concurrent programs*, Proceedings of the 2nd World Congress on Formal Methods (FM 2009) (Ana Cavalcanti and Dennis R. Dams, eds.), Lecture Notes in Computer Science, vol. 5850, Springer, 2009, pp. 256–272, doi:10.1007/978-3-642-05089-3_17.

[WKO13] Björn Wachter, Daniel Kroening, and Joël Ouaknine, *Verifying multi-threaded software with IMPACT*, Formal Methods in Computer-Aided Design (FMCAD 2013) (Barbara Jobstmann and Sandip Ray, eds.), IEEE, October 2013, pp. 210–217, doi:10.1109/FMCAD.2013.6679412.

[WS05] Liqiang Wang and Scott D. Stoller, *Static analysis of atomicity for programs with non-blocking synchronization*, Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2005), ACM, 2005, pp. 61–71, doi:10.1145/1065944.1065953.

[XXZ13] Xinwei Xie, Jingling Xue, and Jie Zhang, *AccuLock: Accurate and efficient detection of data races*, Software: Practice and Experience **43** (2013), no. 5, 543–576, doi:10.1002/spe.2121.

[YCGK08] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby, *Efficient stateful dynamic partial order reduction*, Proceedings of the 15th International SPIN Symposium on Model Checking

Software (SPIN 2008), Springer, 2008, pp. 288–305, doi:10.1007/978-3-540-85114-1_20.

[YGL04] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom, *Memory-model-sensitive data race analysis*, Proceedings of the 6th International Conference on Formal Methods and Software Engineering (ICFEM 2004) (Jim Davies, Wolfram Schulte, and Mike Barnett, eds.), Lecture Notes in Computer Science, vol. 3308, Springer, 2004, pp. 30–45, doi:10.1007/978-3-540-30482-1_11.

[YGLS03] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind, *Analyzing the Intel Itanium memory ordering rules using logic programming and SAT*, Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME) (Daniel Geist and Enrico Tronci, eds.), Lecture Notes in Computer Science, vol. 2860, Springer, 2003, pp. 81–95, doi:10.1007/978-3-540-39724-3_9.

[YGLS04] ———, *Nemos: a framework for axiomatic and executable specifications of memory consistency models*, Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004), IEEE, April 2004, pp. 31–40, doi:10.1109/IPDPS.2004.1302944.

[YWY06] Xiaodong Yi, Ji Wang, and Xuejun Yang, *Stateful dynamic partial-order reduction*, pp. 149–167, Springer, Berlin, Heidelberg, 2006, doi:10.1007/11901433_9.